



“Multi-core Hysteria”: FUD about CRUD?

Andrea O. K. Wright, Chariot Solutions
aok@chariotsolutions.com

These are some of the questions I will answer in this talk:

Since Rails is not currently thread-safe, will Rails fall by the wayside as multi-core technology becomes more prevalent?

What would it take to make Rails thread-safe?

What are the real benefits of thread-safety for Rails developers? What are some alternative concurrency models?

NOTE: This talk was given at RailsConf 2008 on May 31 and June 1. The presentation notes were updated on June 22. <=

Me

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Projects

[Ruby on Rails](#)

[Basecamp](#)

[Highrise](#)

[Backpack](#)

[Campfire](#)

[Ta-da List](#)

[Writeboard](#)

More

[Feed](#)

[Archives](#)

June 06, 2007 11:28

Multi-core hysteria and the thread confusion

New CPUs are growing in cores and not in GHz. That's a tough problem for applications that have been traditionally single-threaded, like games. They have to learn all new techniques and rework their thinking to get the most out of the next-generation platforms.

But the fear of that transition has bled into places where it's largely not relevant, like web-application development. Which has caused quite a few folks to pontificate that the sky is falling for Rails because we're not big on using threads. It isn't.

Multiple cores are laughably easy to utilize for web applications because our problems are rarely in the speed of serving 1 request. The problem is in serving thousands or tens or hundreds of thousands of requests. Preferably per second.

Threads are not the only way to do that. Processes do the job nearly as well with a drop of the complexity. And that's exactly how Rails is scaling to use all the cores you can throw at it.

The 37signals suite is currently using some ~25 cores for the application servers that all the applications have dips on. We'd welcome a 64-core chip any day.

Read more: [A good summary of a discussion on multi-core programming in general.](#)

<http://www.loudthinking.com/posts/7-multi-core-hysteria-and-the-thread-confusion>

This talk was inspired by a blog entry that David Heinemeier Hansson posted last year at around this time called “Multi-core hysteria and the thread confusion.” I had questions about some of his assertions, and I wanted to explore them in depth.

People tend to get somewhat...

MULTI-CORE HYSTERIA

AND THE THREAD CONFUSION

...dramatic when talking about multi-core technology, the extent to which developers will need to completely rewire their thought patterns to take advantage of it, and the percentage of current code bases that will need to be rewritten from the ground up.

There were numerous responses to this blog entry around the Web, ranging from shrill generalizations on reddit to even-handed and well-reasoned blog posts -- but none of them answered my questions to my satisfaction. I hope this presentation will answer some of your questions about the future of Rails as parallelism gets incorporated into more and more system designs.



Photo courtesy, Sun Microsystems

In my abstract, I describe my approach to addressing concerns about Rails and thread-safety as “MythBusters-style.”

Most of you will recognize Adam Savage and Jamie Hyneman, also known as the MythBusters, in this picture.

On their popular TV series, these two special effects experts investigate urban myths and demystify surprising phenomena.

Here they are posing with some of my colleagues from Chariot Solutions at JavaOne a couple of years ago.



Photo: Peter Paugh

Sun held a t-shirt hurling contest that year -- developers were invited to design t-shirt hurling devices that could be used to launch t-shirts into the audience and that used Java technology in some way.

A Chariot team made the t-shirt hurling contest finals, and they were invited to do a demo during JavaOne. Here's a photo of the team in action.

I decided to miss out on all the fun in favor of going to the first RailsConf that year.

I have not been back to JavaOne these last few years, although it sounds like there was some great JRuby content at JavaOne this year.

THANK YOU VERY MUCH

Tony Arcieri	Michael Koziarski	Tali Moreshet	Eero Synatkari
--------------	-------------------	---------------	----------------

Nihn Bui	Hongli Lai	Pratik Naik	Dave Sekowski
----------	------------	-------------	---------------

Erik Cheever	John Lam	Josh Peek	Nick Sieger
--------------	----------	-----------	-------------

Dormando	Tomas Matousek	Dave Poulson	Evan Weaver
----------	----------------	--------------	-------------

Jeremy Kemper	MenTaLguY	Jonathan Rochkind	Ezra Zygmuntowicz
---------------	-----------	-------------------	-------------------

The MythBusters often enlist the help of top experts, and that’s what I did to make sure my information is both current and correct.

When determining whether there’s truth to a rumor the MythBusters track down the source. As you’ll see, in several cases, that’s what I did in more ways than one. I’ll be showing a lot of source code.

Me June 06, 2007 11:28

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Multi-core Hysteria and the Thread Confusion

Projects

[Ruby on](#)

[Rails](#)

[Basecamp](#)

[Highrise](#)

[Backpack](#)

[Campfire](#)

New CPUs are growing in cores and not in GHz....

So, let's take a closer look at David's blog entry. It starts out with: "New CPUs are growing in cores and not in GHz."

Before moving on I want to expand on David's terse description of this industry trend at little bit.

For some time, the industry seemed to be following Moore's Law as if it were a natural law. Moore's Law is often invoked in describing the exponential increases in clock speeds computers ship with on a biyearly basis. But what Gordon Moore actually predicted, back in 1965, had to do with how rapidly the number of components that could economically be placed on a chip would increase.

Cramming more components onto integrated circuits

Integrated circuits will lead to such wonders as home computers--or at least terminals connected to a central computer--automatic controls for automobiles, and personal, portable communications equipment. The electronic wristwatch need only a display to be feasible today.

Gordon E. Moore

Electronics, Vol. 38, No. 8, April 19, 1965

ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

Moore, the co-founder of Intel, made his assertion in an article in Electronics magazine, called "Cramming more components onto integrated circuits."

As an aside, I thought it was interesting that in this same article he predicted that "Integrated circuits will lead to such wonders as home computers."

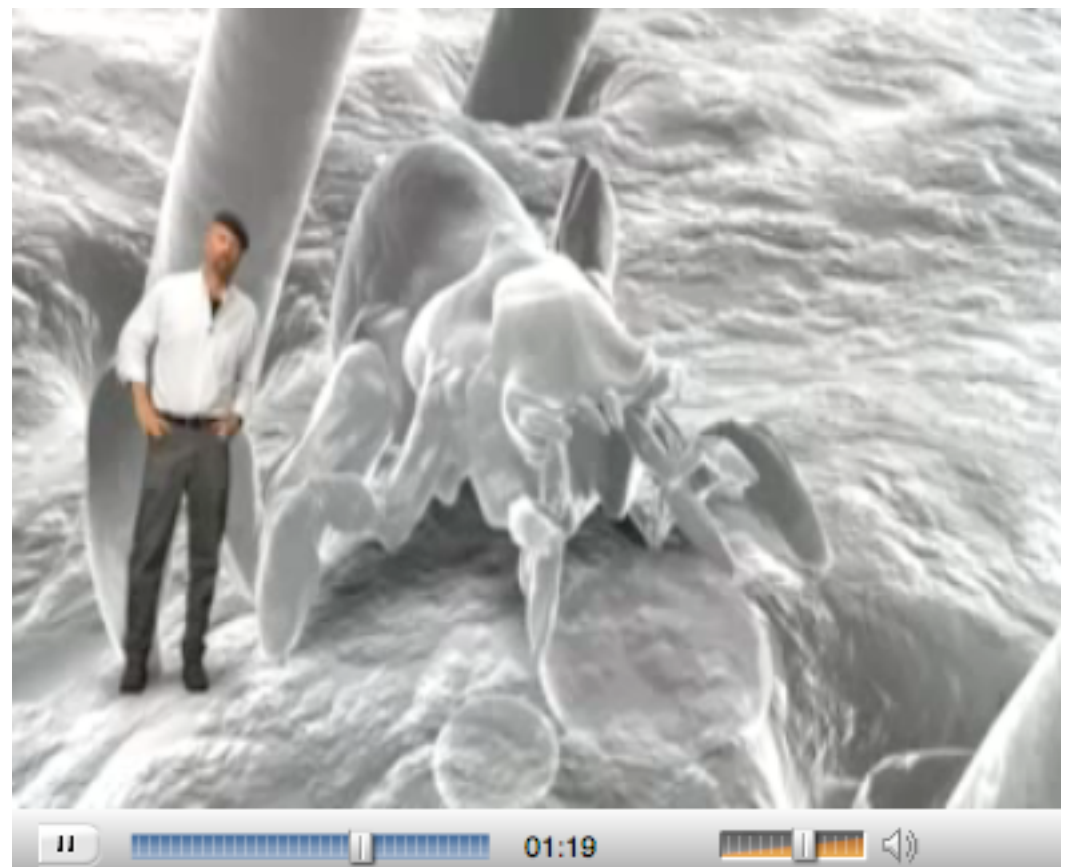
Cramming more components onto integrated circuits



Electronics, Vol. 38, No. 8, April 19, 1965

ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

The article was accompanied by this cartoon, which shows “Handy Home Computers” being sold in a department store.



<http://videos.howstuffworks.com/podtech-networks/2484-mythbusters-guys-talk-chip-size-video.htm>

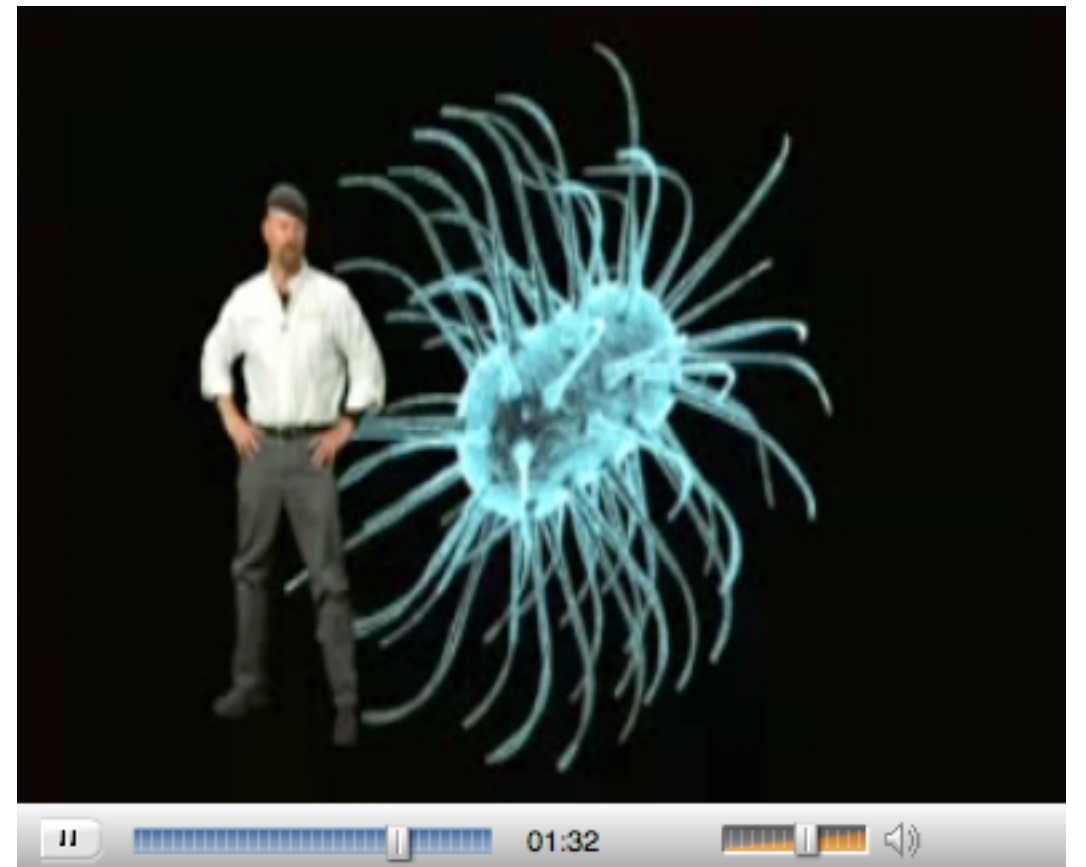
Here are a couple of frames from a short video that Intel commissioned from the MythBusters in conjunction with launching Centrino Duo technology.

They aim to show how quickly transistors have been shrinking -- what it means for it to be possible to fit twice as many transistors on a chip every couple of years or so. In the first frame Adam represents the size of a transistor and Jamie represents the size of a transistor two years later.

There is a correlation between transistor density and clock speed -- and it is that smaller components make it logistically possible to increase physical cache sizes. Reading from a cache closer to a processor is considerably faster than accessing main memory.

For years, developers have been able to take it for granted that when new hardware is released, the old software would run noticeably faster on it.

In the bottom frame, Jamie is showing that a transistor was about the size of a mosquito in 1971.



<http://videos.howstuffworks.com/podtech-networks/2484-mythbusters-guys-talk-chip-size-video.htm>

By 1999 a transistor was small enough to fit inside a red blood cell. The creature in the bottom picture is a bacterium. When the video was made last year, transistors were no bigger than one of its spikes.

While chips are continuing to become more and more transistor-dense, clock speed is starting to plateau. For one thing, faster processors generate intense heat, and today's fans can barely keep up.

To ship computers that have the potential to increase performance, the hardware industry has turned to another solution made possible by decreased component size...

Multi-core

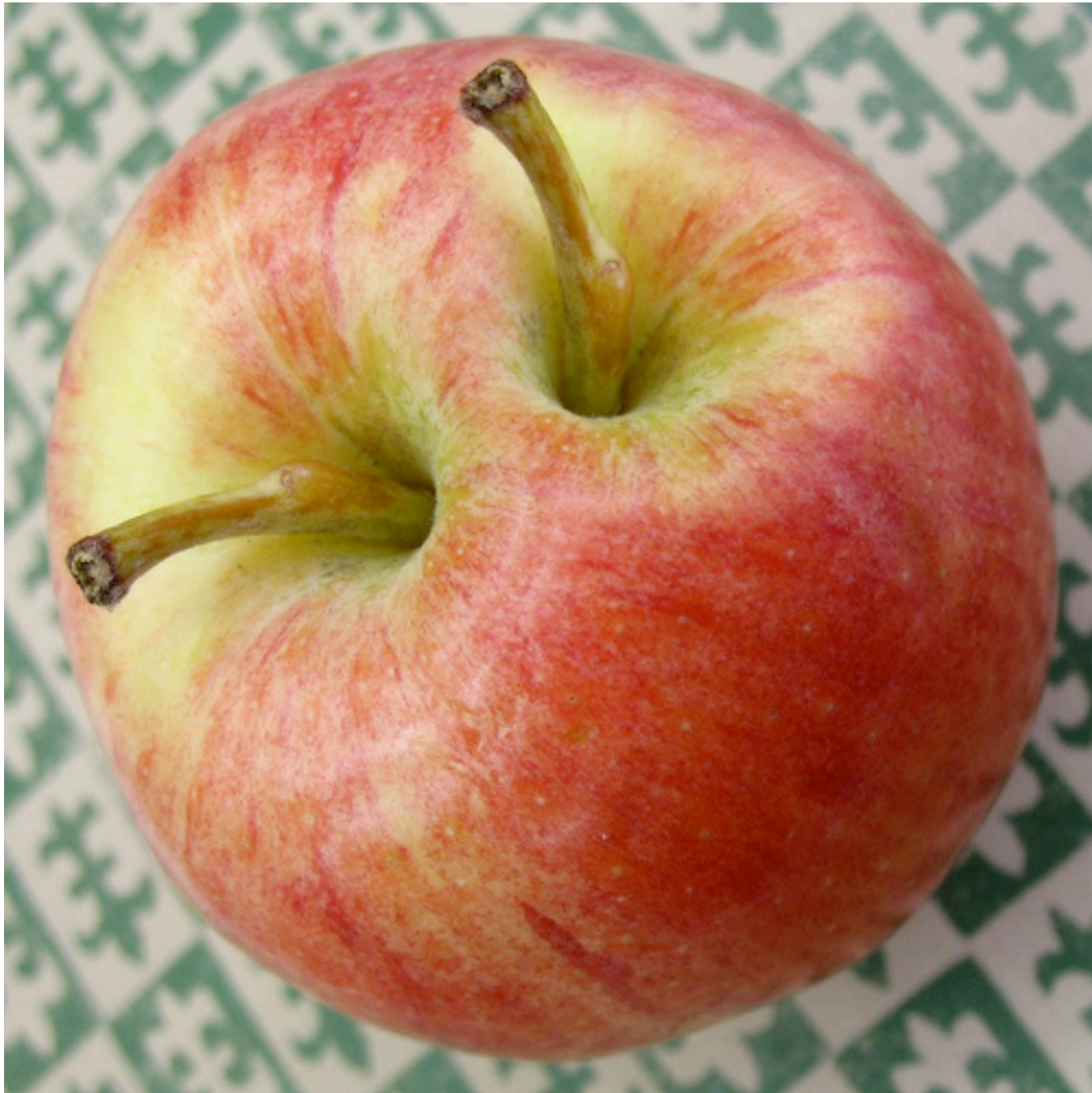


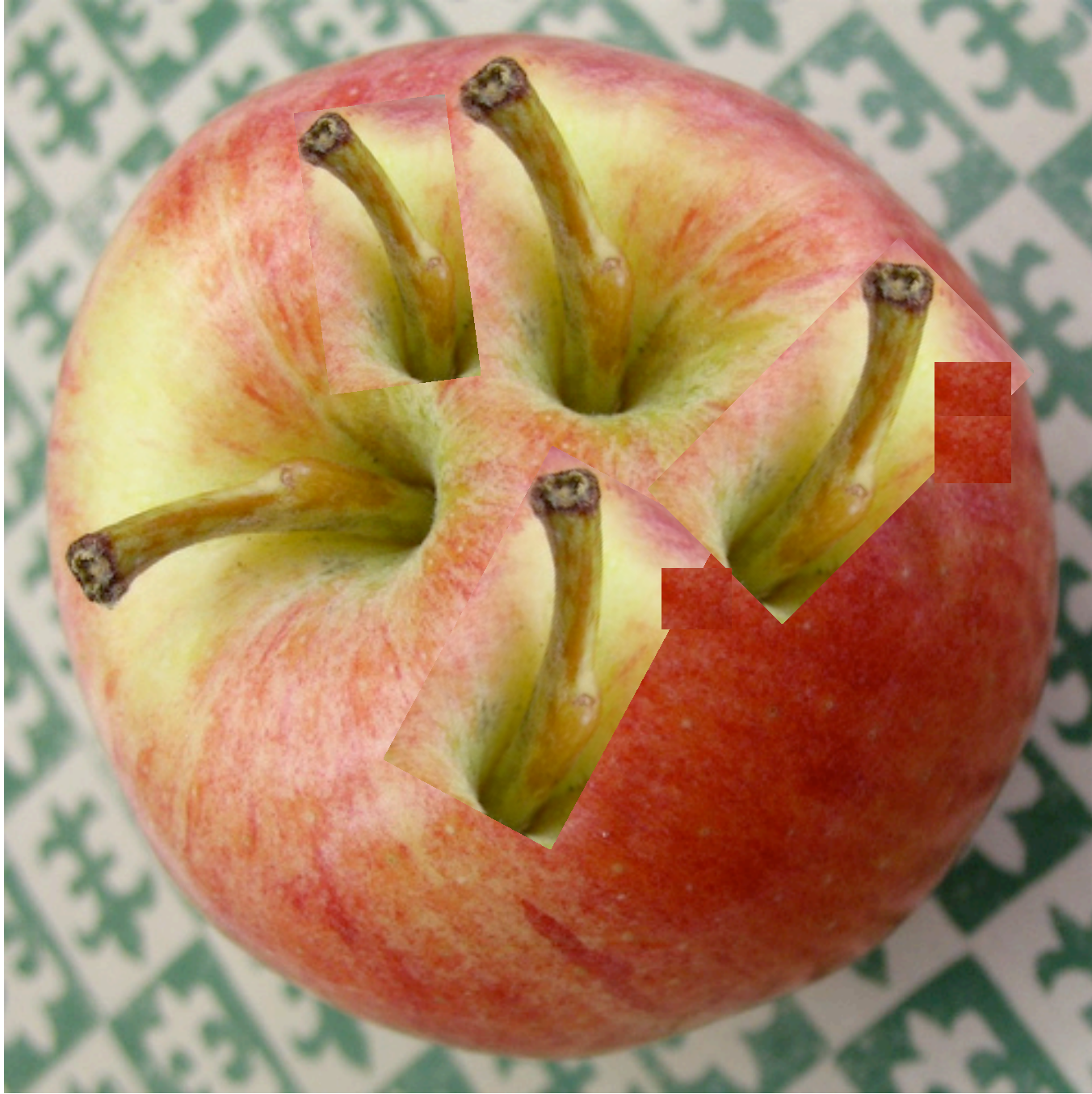
Photo Credit: <http://www.flickr.com/photos/marxalot/406754449/>

...multicore technology. It's becoming possible to ship computers with an increasing number of processing cores built in.

But the only way to leverage more than one core is to design programs with routines that can run in parallel with other routines and\or can process different subsets of data concurrently.

This is where the dire predictions for programmers who are not mastering concurrency concepts come in.

Multi-core



Based on this photo: <http://www.flickr.com/photos/marxalot/406754449/>

And as the number of cores increase, the potential for greater amounts of throughput increases -- but higher numbers of cores introduce additional kinds of complexity that that present challenges to software developers, like subtle bugs having to do with cache mechanics.

In his blog, David specifies video gaming an example of an industry that had no choice but to adapt to the new hardware architecture, but goes on to say...

Me June 06, 2007 11:28

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Projects

[Ruby on](#)

[Rails](#)

[Basecamp](#)

[Highrise](#)

[Backpack](#)

[Campfire](#)

Multi-core Hysteria and the Thread Confusion

[F]ear of ... transition has bled into places where it's largely not relevant, like web-application development. Which has caused quite a few folks to pontificate that the sky is falling for Rails because we're not big on using threads. It isn't.

... “[F]ear of ... transition has bled into places where it's largely not relevant, like web-application development. Which has caused quite a few folks to pontificate that the sky is falling for Rails because we're not big on using threads. It isn't.”

So what are threads, and what are some of the implications of Rails not being thread-safe?

Threads

```
walking = Thread.new do
  100.times do
    puts "walking"
    sleep 1
  end
end

chewing = Thread.new do
  100.times do
    puts "chewing"
    sleep 1
  end
end

walking.join
chewing.join
```

Threads provide a means for achieving concurrency. They provide an alternative to strict sequential processing.

Threads allow a program to walk and chew gum at the same time. Here's the Ruby syntax for a program that does just that.

The join calls at the end ensure that program doesn't exit before the logic in the threads finishes executing.

Threads

```
walking = Thread.new do
```

```
  100.times do
```

```
    puts "w"
```

```
    sleep 1
```

```
  end
```

```
end
```

```
chewing = Thread.new do
```

```
  100.times do
```

```
    puts "c"
```

```
    sleep 1
```

```
  end
```

```
end
```

```
walking.join
```

```
chewing.join
```

```
AOK:~ aok$ ruby walk_chew.rb
```

```
walking
```

```
chewing
```

```
walking
```

```
chewing
```

```
chewing
```

```
walking
```

```
walking
```

```
chewing
```

```
...
```

Here's a portion of the output. We see “walking” interspersed with “chewing” in such a way that it's clear that Ruby did not wait until the first thread was finished before starting to execute the second one.

Threads share memory, so if more than one thread needs to access the same data, the data needs to be protected. Otherwise the results can be unpredictable. Data can morph in unanticipated ways at unexpected times.

HELP! STRANGE BUG IN THREADING: HITTING BACK CAUSES AN ERROR

WEIRD BUG-
DYLIB SYMBOL
LOADING HANGS
ON PTHREAD
SEMAPHORE



<http://lists.apple.com/archives/darwin-development/2002/Dec/msg00175.html>
<http://java-errors.blogspot.com/2008/03/android-developers-re-help-strange-bug.html>
http://blog.cleveland.com/entertainment/2007/07/life_without_the_weekly_world.html

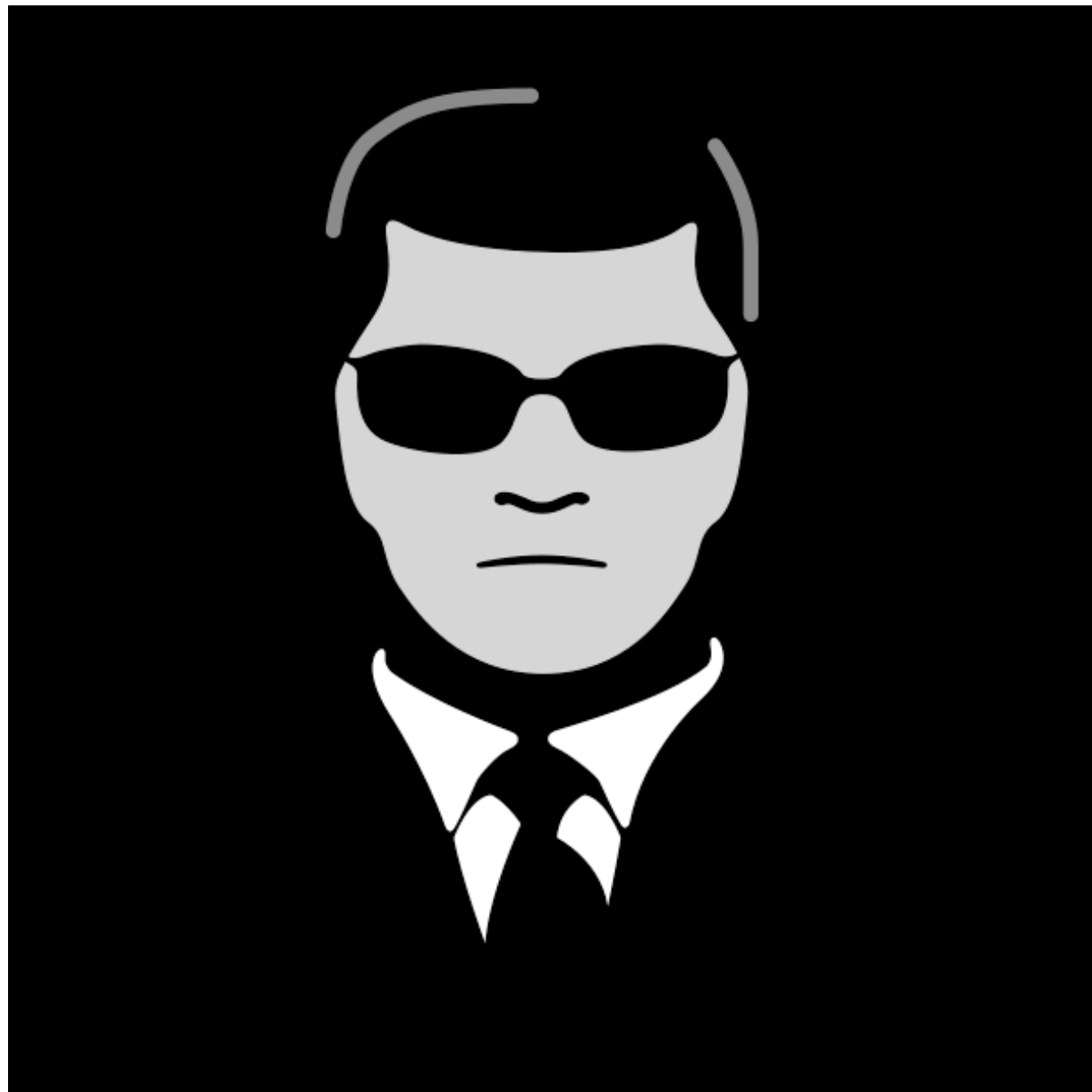
That’s probably why people talk about threading problems as if they are paranormal events, frequently using words like “strange”, “bizarre”, and “weird” .

This article is from the late, great Weekly World News, but the phrases I have set as headlines are taken from the subject lines of actual posts to tech mailing lists.

MUTEX

One way to ensure that only one thread accesses shared data at a time is to use a concurrency primitive known as a Mutex, another weird word that sounds like it might belong in a tabloid headline. Mutex is short for “mutual exclusion lock.”

Men in Black (MIB)



<http://en.wikipedia.org/wiki/Image:MiB.svg>

With all the tabloid sensationalism around the kind of nasty bugs that can surface when threading goes awry, I've been thinking of mutexes in terms of The Men in Black -- pop culture figures who wear black suits and dark sunglasses and show up at UFO sightings to try to keep the witnesses from spreading the story.

In the movie, Men in Black, starring Will Smith and Tommy Lee Jones as agents J and K, respectively, the space aliens and humans live in the same areas, but generally do not cross paths -- not unlike the way multiple threads share the same address space. All is well until a human sees evidence of a space alien. The chaos that ensues is not unlike the programmatic confusion a thread experiences when faced with a value it was not expecting and is not equipped to handle because it was modified by another thread.

The mission of the Men in Black is to keep the humans and space aliens from interacting.

The tabloids figure prominently in the movie. Agents J and K frequently check the tabloids, and not just the articles with headlines that include the word "Space Alien" but also articles with headlines like: "Bat Boy leads Police on a Wild Goose Chase Through 4 states" -- to find out where their services are needed next.

So, how does a mutex keep the threads from accessing the same data at the same time?

Mongrel



```
def run
  ...
  while true
    client = @socket.accept
    ...
    thread = Thread.new(client) {
      |c| process_client(c)
    }
    thread[:started_on] = Time.now
    @workers.add(thread)
    sleep @throttle if @throttle > 0
  end
  graceful_shutdown
  ...
end
```

Here's an abridged version of Mongrel's main loop. When a request comes in, Mongrel creates a new thread for calling "process_client", which in turn calls "process" on the Rails handler, which invokes the Rails dispatcher.

That picture is the newish Mongrel logo.

Note that all the threads spun by Mongrel share the same address space -- and that they all share the Rails instance that Mongrel loads on startup.

There's no thread-safety synchronization code in the Mongrel main loop!

So, what's to stop one Rails request from changing shared data that will confuse another another Rails request?

@@file_exist_cache

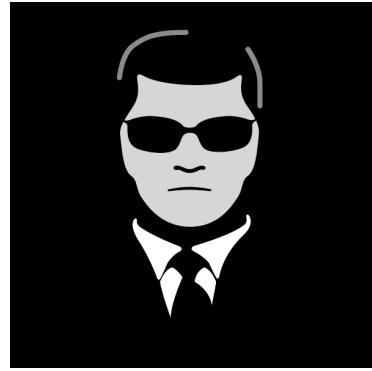
For any of you who may not have a clear idea of specifically why its a problem for multiple threads running Rails requests to share the same address space or specifically what I mean by “shared data that must be protected”, I’m going to go into a lot of detail about where Rails is not thread-safe in short order. For now, here’s something concrete:

Class variables constitute a “red flag” of sorts when analyzing code for what needs protection if its going to be shared among threads.

This is a class variable used by the Rails framework to cache code for the module that contains helpers like `image_tag` and `javascript_tag`. Picture thread 1 and thread 2 both looking at this variable. If thread 1 changes its value, the new value becomes the current value for thread 2 -- where it may not be accurate.

Mongrel RailsHandler

```
class RailsHandler < Mongrel::HttpHandler
  ...
  def initialize(dir, mime_map = {})
    @guard = Mutex.new
    ...
  end
  def process(request, response)
    ...
    cgi = Mongrel::CGIWrapper.new(request, response)
    ...
    @guard.synchronize {
      ...
      Dispatcher.dispatch(cgi,
        ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS,
        response.body)
    }
  end
end
```



Enter: the Men in Black!

Here's the Mongrel Rails handler that's packaged with Mongrel. That's where the mutex is applied. Although multiple Mongrel threads call "process" on this handler in parallel, the mutex around the call to the Rails dispatcher ensures that only one thread has exclusive access to the Rails code base at a time.

In order for a Rails app to service multiple concurrent requests with Mongrel, it's necessary to run multiple instances of Mongrel, colloquially referred to as "a pack of Mongrels."

Running multiple Mongrels requires more system resources than running a single instance that can be shared by all the requests. This is one of the main complaints brought up by those who advocate thread-safety for Rails.

Userspace Threads Green Threads

Native Threads Kernel Threads OS Threads

There are a few other things that I'd like to mention about threads before we go back to DHH's blog entry, and those include the difference between userspace threads (also known as green threads) and OS threads (which are also referred to as kernel threads or native threads).

Native Threads are managed by the kernel, which can run them in parallel, routing them to multiple cores on machines with multicore support.

Userspace threads are managed by a virtual machine. They are not visible to the kernel, which therefore can't direct them to different cores for processing. They can appear to run simultaneously because the VM time slices between them so quickly.

Userspace Threads Green Threads

Ruby 1.8x

Native Threads Kernel Threads OS Threads

Threads in Ruby 1.8x are userspace threads.

This is one of the reasons why thread-safety has not been a priority for Rails.

Userspace Threads Green Threads

Ruby 1.8x

Native Threads Kernel Threads OS Threads

Ruby 1.9*

Ruby 1.9 has native thread support, but I added an * because there is a global lock in place that prevents more than one thread from executing at a time. One of the reasons for this is that the not all of the Ruby stdlib libraries are thread-safe.

Userspace Threads Green Threads

Ruby 1.8x

Native Threads Kernel Threads OS Threads

**Ruby 1.9*
JRuby
IronRuby**

JRuby leverages Java’s mature memory model and strong support for native threading.

Likewise, IronRuby leverages the CLR’s support for native threading.

Userspace Threads Green Threads

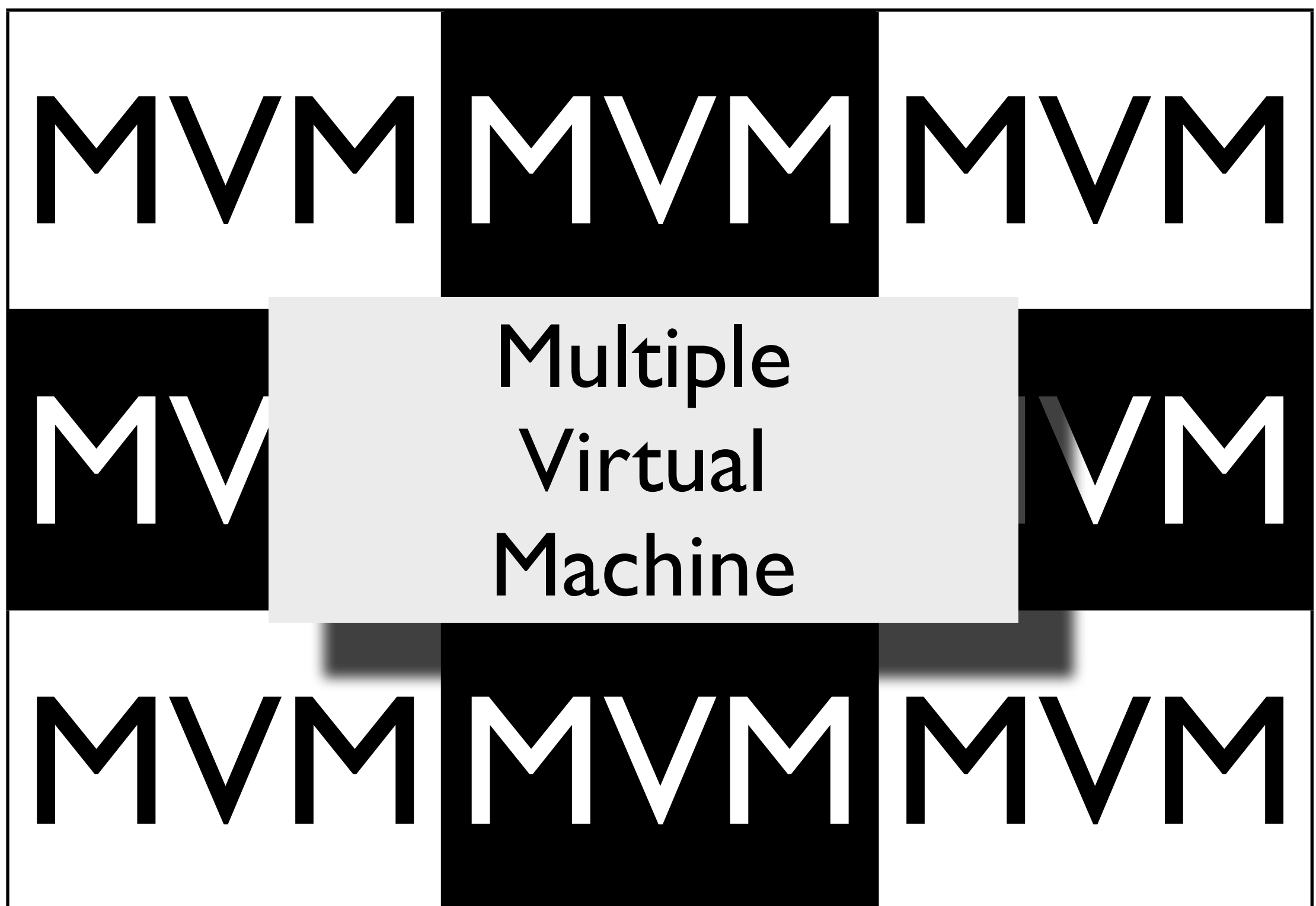
Ruby 1.8x
Rubinius*

Native Threads Kernel Threads OS Threads

Ruby 1.9*
JRuby
IronRuby

For Rubinius, the threading API is implemented to use green threads.

Rubinius is starred with a “*” because the plan is for the threading implementation to switch to use native threads at some point -- and also because Rubinius does have some native threading support currently in that it’s possible to spawn multiple Rubinius VMs and map each to a native thread.



JRuby also has multiple VM (MVM) support, and MVM support is under consideration for Matz Ruby as well. By Multiple VM support I mean the ability for multiple VMs to share a single interpreter.

Developers representing the core teams from these and other Ruby implementations are working together on researching how to best use this capability and on a common MVM communication protocol.

VMs often come up in discussions about parallelism because they have the potential to be mapped to native threads.

MVM capability plays a part in the Rails deployment strategies offered by several Ruby implementations because VMs can run in parallel, and also because they can provide a degree of isolation or sandboxing for code that runs inside them -- while at the same time reducing memory requirements by sharing bytecode.

Rubinius : Friday 2008/04/11

01:47:23 evan

it's software people, nothing is impossible.

Evan Phoenix

As Rubinius lead developer Evan Phoenix said on the Rubinius IRC channel last month during a discussion about VM architecture: it's software people, nothing is impossible.

Me June 06, 2007 11:28

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Projects

[Ruby on](#)

[Rails](#)

[Basecamp](#)

[Highrise](#)

[Backpack](#)

[Campfire](#)

Multi-core Hysteria and the Thread Confusion

Multiple cores are laughably easy to utilize for web applications because our problems are rarely in the speed of serving 1 request. The problem is in serving thousands or tens or hundreds of thousands of requests. Preferably per second.

Anyway, in the blog entry I was telling you about, David goes on to say: “Multiple cores are laughably easy to utilize for web applications because our problems are rarely in the speed of serving 1 request. The problem is in serving thousands or tens or hundreds of thousands of requests. Preferably per second.”

So while he’s not saying that there are never any scenarios where tapping multiple CPUs simultaneously would come in handy, he’s saying here that by far the most predominant use case for multicore technology in the Web application space is handling concurrent requests.

Request handling is what’s known as an “embarrassingly parallel” task -- one where multiple cores can easily be exploited because the requests are typically independent of one another. Sequencing and data distribution are non-issues.

Me June 06, 2007 11:28

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Projects

[Ruby on](#)

[Rails](#)

[Basecamp](#)

[Highrise](#)

[Backpack](#)

[Campfire](#)

Multi-core Hysteria and the Thread Confusion

Threads are not the only way to do that. Processes do the job nearly as well with a drop of the complexity. And that's exactly how Rails is scaling to use all the cores you can throw at it.

DHH continues: "Threads are not the only way to do that. Processes do the job nearly as well with a drop of the complexity. And that's exactly how Rails is scaling to use all the cores you can throw at it."

So, we're going to be considering the question: Do processes do the job of exploiting multiple processors well enough -- and is there really a net drop in complexity with multi-process deployments?

Earlier we looked at the Mongrel source and discussed why it's necessary to run multiple Mongrels in order to service multiple Rails requests. I didn't say it explicitly at the time but those requests are each running in their own processes. I'm going to refer to deployments like that as "multi-process" strategies or "processed-based" -- as opposed to threaded.

Process

has_one :thread

Process

has_many :threads

What is the relationship between threads and processes?

A process is allotted its own physical system resources.

A thread belongs to a process.

A process has at least one thread, usually referred to as the “main thread”, but can spawn multiple threads.

A thread is sometimes referred to as a lightweight process. Because it shares its parent process’s address space, it can be started from within a process with very little in the way of additional system resources or the overhead associated with doling them out.

Conversely, spawning a subprocess is more resource-intensive than spawning a thread.

Processes

```
walking = fork do
  100.times do
    puts "walking"
    sleep 1
  end
end

chewing = fork do
  100.times do
    puts "chewing"
    sleep 1
  end
end

Process.wait(walking)
Process.wait(chewing)
```

As DHH suggested, processes represent another way to programatically walk and chew gum at the same time, and processes can be routed to multiple cores.

One way to create a subprocess in Ruby is to call `fork`. Unlike threads, subprocesses have their own address spaces. Subprocesses each start out as replicas of the parent process, but when anything changes, those changes are written to the child process's own address space.

Processes can only communicate with each other through Interprocess Communication (IPC) protocols, like pipes. Parent and sibling processes even need these protocols to communicate with each other.

The `wait` calls ensure that the child processes will finish before the parent exits. To indicate to the system that the parent doesn't plan to wait for the child to finish, use `Process.detach`.

If the parent process exits before the child there will be ...

ZOMBIE

PROCESSES

... zombie processes.

Speaking of zombie processes, it's not uncommon for conversations about Rails and thread-safety to come around to the topic of zombie processes eventually.

I know of Rails deployments that have been plagued by zombie processes taking up enough system resources to make the system noticeably slower or even using them all up.

It's not actually a Rails problem or a problem with most process-based deployments. It has to do specifically with an older version of apache's mod_fcgi.

But the zombie process reference has come to represent general dissatisfaction with aspects of some common multiprocess deployment scenarios.

For example, when Charles Nutter of the JRuby project blogged relatively recently about a JRuby alternative to typical process-based deployments with Apache, he wrote, in part, "no more zombie processes."

Memory



<http://www.flickr.com/photos/56871332@N00/197941804/sizes/m/>

Ease



<http://blog.provokat.ca/en/index.php/?archives/364-The-Rube-Goldberg-effect.html>

Speed



<http://www.flickr.com/photos/airport/65934250/>

These are the main common complaints with common multi-process deployment schemes, like Mongrel Cluster, Apache\FastCGI or lighttpd\FastCGI.

Memory: They require too much of it. The Rails framework typically needs to be loaded in each each process.

Ease: There are too many moving parts involved. The more processes involved, the harder a deployment is to maintain. Setup is also a hassle.

Speed: This is about latency. With a single shared instance launch time is not as much of an issue.

Are there good deployment options that don't involve threading?

That's good.

That's bad.

The children found themselves face to face with a bear.

That's bad.

It was a baby bear cub.

That's good.

The cub's mother was nearby though.

That's bad.

The mother bear was partial to lemon squares, and the children happened to have packed some for lunch.

That's good.

A lot has been changing in the concurrency space since I proposed this talk at the end of last year. As I was preparing for this talk, I often thought about those folk tales or clown routines based around the phrases “That's good” and “That's bad”.

That's good.

That's bad.

Rails is not thread-safe, and can't be deployed on threaded servers without a lock.

That's bad.

But you can effectively service multiple concurrent requests by spawning multiple processes with a load balancer.

That's good.

Memory requirements are high though, because Rails must be loaded for each process.

That's bad.

No need to share Rails if you fork after Rails is loaded.

That's good.

But problems w\the way Ruby handles GC reduce any memory savings.

That's bad.

Hongli Lai of Phusion created a patch to address those GC issues, and it's freely downloadable.

That's good.

This slide shows how “That's good\That's bad” played out with respect to the topic at hand, summarizes some of what I've talked about, and foreshadows some of what I'm going to be talking about.

At this point I'd like to analyze Rails from a thread-safety perspective. That way, when the phrase “not thread-safe” comes up in the rest of the talk it won't be so much of broad generalization. You'll be able to picture some of the particulars.

We're going to look at ActiveRecord separately, after we look at some of the other Rails packages. That's because ActiveRecord is frequently used outside of Rails, for example as the ORM for a Merb application. So it makes sense to consider whether ActiveRecord can be run in a threaded environment apart from whether there are thread-safety issues with its Rails integration.

Application Information

[Google Code Home](#) > [Google Summer of Code](#) > [Organization Information](#) > Application Information

Title	Rails Thread Safety
Student	Joshua Peek
Mentor	Michael Alan Koziarski

Abstract

The goal of the project would be to make the Rails framework as thread safe as possible.

Source Browser

Commits

Wiki

Network

thread_safe

all branches

all

josh / rails

fork

watch

download

Fork of rails/rails

Description: Ruby on Rails

Homepage: <http://rubyonrails.org>

Clone URL: <git://github.com/josh/rails.git>

Repo Search:

Code

Go

The big news on the Rails and thread-safety front is that the goal of Rails committer Josh Peek’s Google Summer of Code Project is to focus on Rails and thread safety. Rails core team member Michael Koziarski is his mentor for the project.

The goal of the project is actually to make Rails thread-safe in production mode. What would be the purpose of thread-safety for development mode? Development mode depends on being able to reload files before every request, so there’s a short feedback loop. Maybe some day after its safe to run multithreaded Rails in production mode, someone will work on a collaborative development environment that keep developers from overwriting each other’s changes.

There has been more interest in resolving thread-safety issues in Rails as the concurrency polices of the different Ruby implementations have been evolving to support native threads.

Josh will be doing a lot of analysis and design work. Testing will be involved to some extent, but its important to remember that you can’t guarantee that something is thread-safe just because they have not shown up in testing.

You can follow Josh’s progress on the thread_safe branch he created on github.

He’s just now getting started with the project, but he’s already posted some interesting code.

UPDATE: Josh has discontinued his thread_safe branch in favor of creating multiple branches that each address a single issue related to thread-safety. For example, the his preload branch (<http://github.com/josh/rails/tree/preload>) includes the fixes for the Dependencies-loading issue described on the next slide. It will be easier to merge the changes into Edge Rails as they are ready to be checked in, as opposed to trying to merge in all the changes from a monolithic thread_safe branch at the end of the summer. <=

Ticket #9155 (new defect)
**The Dependencies system
isn't thread safe**
Reported
by: Catfish

some_model.rb :

```
class SomeModel
  sleep 0.5    # arbitrary delay to make
  def self.is_fully_loaded
    true
  end
end
```

executing code :

```
threads = []
threads << Thread.new do
  SomeModel.is_fully_loaded
end
threads << Thread.new do
  SomeModel.is_fully_loaded
end
threads.each(&:join)
```

<http://dev.rubyonrails.org/ticket/9155>

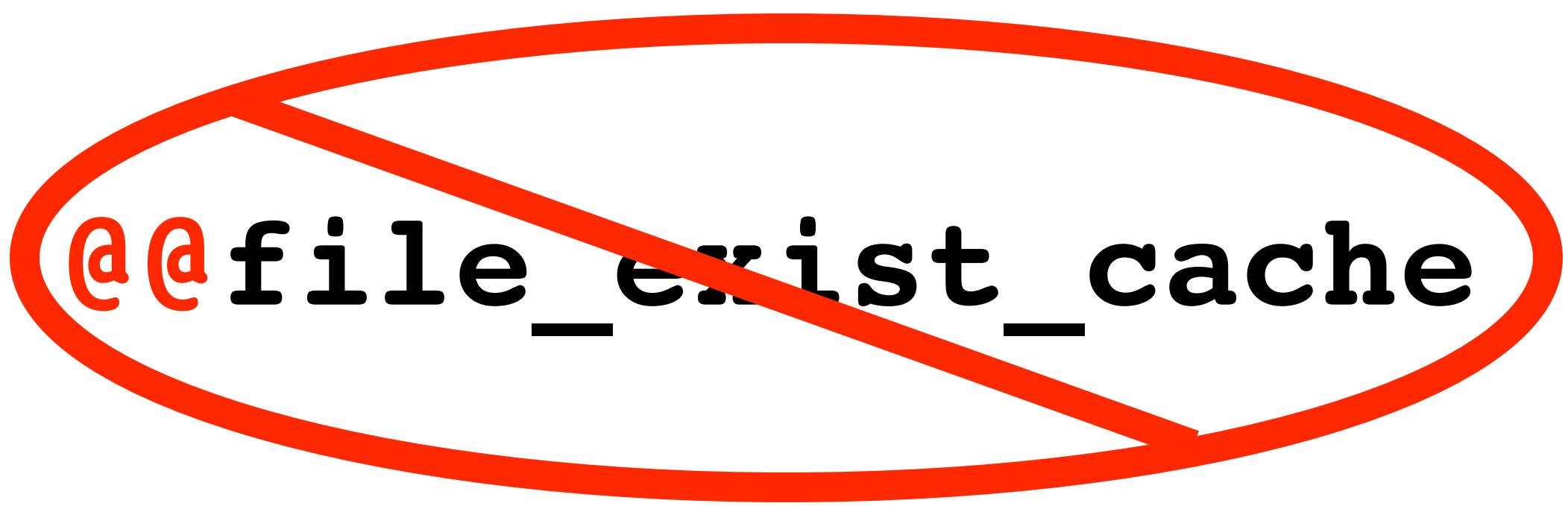
One of the first things he will be tackling is the Dependencies loading system.

The Dependencies module is the magic that frees you from needing to use explicit “require” statements in your code. When Rail encounters a constant it doesn’t know about in the course of running your code, in development mode and in production mode -- it’s `const_missing` from the Dependencies module that triggers a search through all load paths your Rails environment knows about, and loads the necessary files.

Catfish does a good job of demonstrating the problem when Rails is running in multi-threaded mode in the ticket he posted in the old Rails Trac instance.

He created a model called `SomeModel` with a method called `is_fully_loaded` and describes a scenario that’s likely to play out if you run his sample code a few times. The first thread’s call to `is_fully_loaded` on `SomeModel` initiates the file loading by way of the `const_missing` code. While the file is in the process of loading, the second thread calls `is_fully_loaded`. It’s possible for the second thread to call `is_fully_loaded` after the constant `SomeModel` is loaded, but before the `is_fully_loaded` method is defined. Class loading in Ruby is not atomic.

The solution Josh is working on is a preload module, which takes care of pre-loading all the constants before Rails processes any requests in production mode.



~~@@file_exist_cache~~

```
FILE_EXIST_CACHE=  
ActiveSupport::Cache::LookupStore(:memory_store)
```

One of the other things Josh is doing is experimenting with leveraging the cache store classes that come with ActiveSupport for the kinds of caches that are typically stored in class variables today.

UPDATE: Josh has put this experiment on hold for now to investigate whether any of these caches can be populated and frozen at initialize time, as an alternative to a solution that involves locking. As you will see in the next few slides, the initial solution involved implementing synchronization for the cache store's read, write and delete routines. **<=**


```
class Store
  ...
  def threadsafe!
    @mutex = Mutex.new
    self.class.send :include, ThreadSafety
    self
  end

  def read(key, options = nil)
    log("read", key, options)
  end

  def write(key, value, options = nil)
    log("write", key, options)
  end

  def delete(key, options = nil)
    log("delete", key, options)
  end

  def delete_matched(matcher, options = nil)
    log("delete matched", matcher.inspect, options)
  end
  ...
end
```

Here's the class that all the cache stores (file_store, mem_cache_store, etc) extend.

The Rails core team plans to support the option of making the Store class thread-safe but the exact mechanism has not yet been determined. Currently there's a method called "thread-safe"...

```

class Store
  ...
  def threadsafe!
    @mutex = Mutex.new
    self.class.send :include, ThreadSafety
    self
  end

  def read(key, options = nil)
    log("read", key)
  end

  def write(key, value, options = nil)
    log("write", key, value)
  end

  def delete(key, options = nil)
    log("delete", key)
  end

  def delete_matched(matcher, options = nil)
    log("delete matched", matcher)
  end

  ...
end

```

```

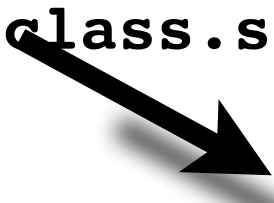
module ThreadSafety
  def read(key, options = nil)
    @mutex.synchronize { super }
  end

  def write(key, value, options = nil)
    @mutex.synchronize { super }
  end

  def delete(key, options = nil)
    @mutex.synchronize { super }
  end

  def delete_matched(matcher, options = nil)
    @mutex.synchronize { super }
  end
end

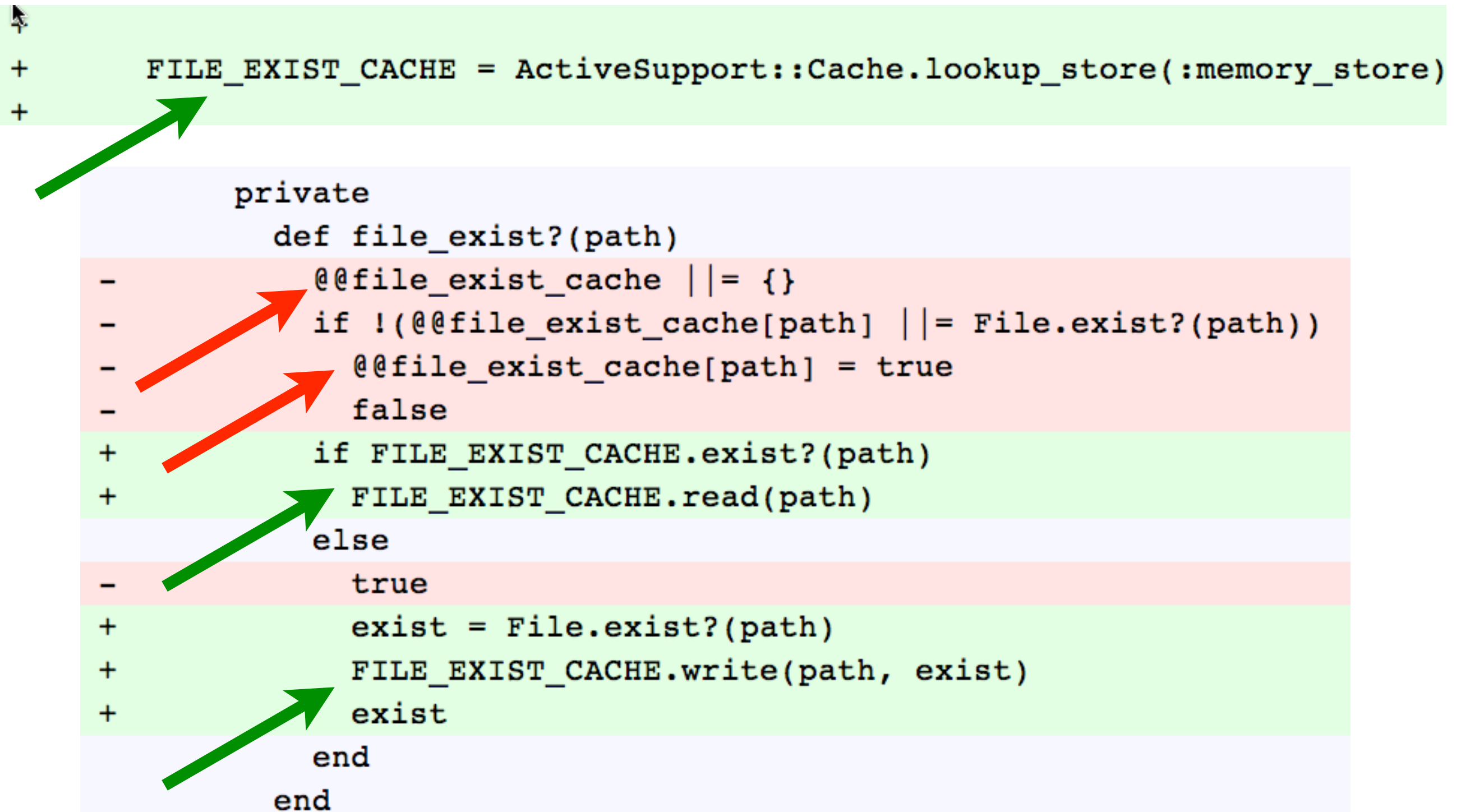
```



<http://dev.rubyonrails.org/ticket/9155>

...loads a module that overrides several Store methods with synchronized versions of those methods. This method needs to be called before Rails processes any requests. There's currently no way to enforce that, but in the future, when ActiveSupport is officially thread-safe, it may be called from the initialize method in Store, based on whether Rails is running in multi-threaded mode.

```
+ FILE_EXIST_CACHE = ActiveSupport::Cache.lookup_store(:memory_store)
+
private
  def file_exist?(path)
-   @@file_exist_cache ||= {}
-   if !(@@file_exist_cache[path] ||= File.exist?(path))
-     @@file_exist_cache[path] = true
-     false
+     if FILE_EXIST_CACHE.exist?(path)
+       FILE_EXIST_CACHE.read(path)
    else
-     true
+     exist = File.exist?(path)
+     FILE_EXIST_CACHE.write(path, exist)
+     exist
  end
end
```



<http://github.com/josh/rails/commit/300426e21e92a21286bd99165d76866b0dfec359>

Here's an example where he has substituted a class variable with an ActiveSupport::Cache-based store.

It does some caching in AssetTagHelper, the module that contains javascript_tag, image_tag and friends.

The mechanism for making ActiveSupport::Cache thread-safe has not been determined yet.

The premise is really really simple to understand:

1. ★ Your tests should pass.
2. ★ Break your code.
3. ★ Now they should fail.

josh / thread_heckler

Heckle sees this code and mutates every colored element, just to make sure you checked them.

`if` becomes `unless`, `calls` get replaced, `numbers` get changed, `assignments` get changed, etc.

```
if month > 12 then
  if month % 12 == 0 then
    year += (month - 12) / 12
    month = 12
  else
    year += month / 12
    month = month % 12
  end
end
```

http://github.com/josh/thread_heckler/tree/master <http://ruby.sadi.st/Heckle.html>

In conjunction with his Thread safety branch he started working on Thread Heckle, an experimental version of Heckle customized for trapping thread-safety violations.

The text with the black background is taken from the Heckle page at the Ruby Sadist site (<http://ruby.sadi.st/Heckle.html>).

In case you are not familiar with Heckle, it's a plugin that mutates your code on the fly and runs your tests to determine how meaningful your tests are. For example, it might dynamically replace strings with random characters before running tests. If your tests still pass after a string is mutated, you clearly don't have tests that care about the value of that string. Heckle even replaces conditional constructions like `if`, `while`, `unless` or `until`.

Heckle is able to identify branching constructs and provides a mechanism for generating a large number of different execution paths.

Josh has been thinking about the best way to use Heckle's infrastructure to simulate concurrency edge cases. He's added code to thread tests when Heckle runs them, as well as a mechanism for adding `Thread.pass` calls at random times. `Thread.pass` directs the thread-scheduler to switch to another thread.

UPDATE: Josh has discontinued his thread_heckler experiment. Traversing all the branching constructs for a project the size of Rails was taking too long. And this tool was not catching threading issues that were not fairly obvious. `<=`

While it's big news that Josh is going to focus on Rails and thread-safety, the Summer of Code does not mark the first time Rails developers have addressed thread-safety

```
Time.zone="Eastern Time (US & Canada)"
```

Core committer Geoff Buesing added time zone support features to Rails for the 2.1 release. With the code on this slide, the Rails-wide time zone becomes Eastern Time. For his initial check-in, zone was an class instance variable for the Time class, and therefore shared among threaded requests.

trunk/activesupport/lib/active_support/core_ext/time/zones.rb

```
module ClassMethods
```

```
  attr_reader :zone
```

```
  def zone
```

```
    Thread.current[:time_zone]
```

```
  end
```

```
  def zone=(zone)
```

```
    @zone = get_zone(zone)
```

```
  def zone=(time_zone)
```

```
    Thread.current[:time_zone] = get_zone(time_zone)
```

```
  end
```

<http://dev.rubyonrails.org/changeset/8718>

But soon after that he decided it was worth his while to make the Time Zone handling thread-safe, and instead of using a class instance variable for storage, he used a thread-local variable. Here's part of the change-set. The syntax for thread-local variables is hash-like, as if Thread.current was the hash identifier. Here you can see how to set a thread-local variable and also how to retrieve its value.

trunk/activereource/lib/active_resource/connection.rb

```
def http
```

```
  unless @http
```

```
    @http = Net::HTTP.new(@site.new(@site.host,  
                                   @site.port)
```

```
    @http.use_ssl = @site.is_a?(URI::HTTPS)
```

```
    ...
```

```
  end
```

```
  @http
```

```
  http = Net::HTTP.new(@site.new(@site.host,  
                                   @site.port)
```

```
  http.use_ssl = @site.is_a?(URI::HTTPS)
```

```
  ...
```

```
  http
```

```
end
```

<http://dev.rubyonrails.org/changeset/8167>

Here's part of a change set that makes ActiveRecord more thread-safe. It replaces the code that caches the http object in an instance variable for the Connection class (which is shared among requests) with code that creates a new http object whenever the resource needs to communicate with the remote server.

It might seem that the main thread-safety issue here is that the site port or host could be different for different threads, but in practice neither the application code or the framework code change this value once it is set.

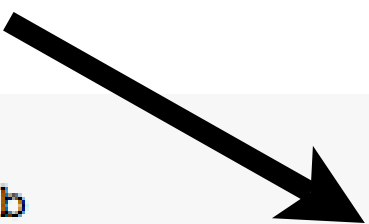
```
% cd /tmp/sample
% cat > requests.rb
module Webservice
  class Book < ActiveRecord::Base
    self.site = "http://localhost:3000"
  end
end

before = Book.count

Thread.abort_on_exception = true
threads = []
100.times do
  threads << Thread.new do
    Webservice::Book.create(:title => "Ruby")
  end
end
threads.each(&:join)
```

<http://dev.rubyonrails.org/changeset/8167>

Here's the trac ticket, which was initiated by kou. If 100 books are created using the ActiveRecord API, invariably



```
puts Book.count - before
% script/runner requests.rb
/usr/lib/ruby/1.8/net/protocol.rb:176:in `write0': You have a nil object when
You might have expected an instance of Array.
The error occurred while evaluating nil.+      from /usr/lib/ruby/1.8/net/pr
      from /usr/lib/ruby/1.8/net/protocol.rb:166:in `writing'
      from /usr/lib/ruby/1.8/net/protocol.rb:150:in `write'
      from /usr/lib/ruby/1.8/net/http.rb:1540:in `send_request_with_body'
      from /usr/lib/ruby/1.8/net/http.rb:1525:in `exec'
      from /usr/lib/ruby/1.8/net/http.rb:1048:in `request'
      from /usr/lib/ruby/1.8/net/http.rb:845:in `post'
      from /tmp/sample/vendor/rails/activeresource/lib/active_resource/conn
      ... 17 levels...
      from script/runner:3:in `eval'
      from /tmp/sample/vendor/rails/railties/lib/commands/runner.rb:45
      from script/runner:3:in `require'
      from script/runner:3
```

'script/runner requests.rb' should output '100'.

<http://dev.rubyonrails.org/changeset/8167>

...this error shows up when a method is called on a “nil object” in the net library code. It turns out that the net::http library is not thread-safe. Behind the scenes its using instance variables to track its buffer state, and it gets confused when these values are modified in different threads.

Putting a mutex where the http object is used in the Rails code would be one way to fix the problem, but it was decided to go with instantiating a new http object for each transaction.

```
class CGI
  ...
  def env_table
    ENV
  end
  ...
end
```

I want to address `cgi.rb` and thread-safety because I've heard people cite dependence on `cgi.rb` as one of the reasons why Rails is not thread-safe. Rails uses the CGI web protocol as part of its request processing.

There are thread-safety issues in `cgi.rb`, but Rails does not use `cgi.rb` in a non-threadsafe manner.

This top snippet of code is from `cgi.rb`, and you can see that it uses an environmental variable that could not be shared across requests without a Mutex: `ENV`.


```
class CGI
  ...
  def env_table
    ENV
  end
  ...
end

module Mongrel
  class CGIWrapper < ::CGI
    ...
    def env_table
      @request.params
    end
    ...
  end
end
```

However Rails does not use the cgi.rb ENV variable.

It uses a CGIWrapper with env_table overridden to reference the @request instance variable. The CGI::Wrapper redefinition of env_table is the bottom code snippet.



Ezra Gets Rails® On Rack

POST BY PETER COOPER

[Permanent Link](#)

|  [Del.icio.us](#)

|  [Cosmos](#)



<http://www.rubyinside.com/rails-on-rack-872.html>

Speaking of `cgi.rb` -- as it turns out, in just the last couple of months, Ezra Zygmuntowicz, who most of you probably know, is the creator of Merb and a co-founder of EngineYard added support for the Rack web server interface to Rails, and removed references to `cgi.rb` in the process. He did the work in his own github branch, but it was with the understanding that he would merge the changes into EdgeRails not long after the imminent 2.1 release. Ezra was able to make some nice improvements, but he didn't make changes that correct thread-safety violations, explicitly.

UPDATE: Edge Rails is now Rack-enabled. Ezra's Rack-related changes were merged in, but the CGI dependencies will not be removed until the Rack\Rails integration code is tested more. For more details, read this Rails core mailing list thread: http://groups.google.com/group/rubyonrails-core/browse_thread/thread/237bb20d25ef7e57/141d0a9d825a7cb9?lnk=gst&q=Rack#141d0a9d825a7cb9. <=

There is one change that Ezra made to the version of Rails in his branch that's especially relevant to this talk: he changed the placement of the Mutex so that it protects less of the Rails codebase.

Mongrel RailsHandler

```
class RailsHandler < Mongrel::HttpHandler
  ...
  def initialize(dir, mime_map = {})
    @guard = Mutex.new
    ...
  end

  def process(request, response)
    ...
    cgi = Mongrel::CGIWrapper.new(request, response)
    ...
    @guard.synchronize {
      ...
      Dispatcher.dispatch(cgi,
        ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS,
        response.body)
    }
  end
end
```

I want to clarify some things about the lock and its placement, for those of you who are going to look at the Rails source to see for yourself.

First of all, here's the code from the Rails handler packaged with the most recent version of Mongrel. We looked at this a little while ago.

It wraps the entire “dispatch” call, which includes dispatcher callbacks, route recognition, and controller instantiation.

Changeset 8488

Timestamp: 12/27/07 (5 months ago)

Author: bitsweat

Message: Introduce native mongrel handler and push mutex into dispatcher.

[trunk/actionpack/lib/action_controller/dispatcher.rb](#) (2 diffs)

[trunk/railties/CHANGELOG](#) (1 diff)

[trunk/railties/lib/commands/server.rb](#) (2 diffs)

[trunk/railties/lib/commands/servers/new_mongrel.rb](#)

[trunk/railties/lib/rails/mongrel_server](#)

[trunk/railties/lib/rails/mongrel_server/commands.rb](#)

[trunk/railties/lib/rails/mongrel_server/handler.rb](#)

Core team member Jeremy Kemper moved the lock from the Mongrel handler source to the Rails source, shortly after Rails 2.0 was released.

He did not change the Mongrel code base to remove the lock.

/railties/lib/rails/mongrel_server/handler.rb

```
class RailsHandler < Mongrel::HttpHandler
  ...
  def initialize(dir, mime map = {})
    @guard = Mutex.new
    ...
  end

  def process(request, response)
    ...
    cgi = Mongrel::CGIWrapper.new(request, response)
    ...
    @guard.synchronize {
      ...
      Dispatcher.dispatch(cgi,
        ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS,
        response.body)
    }
  end
end
```

He introduced this Mongrel handler without a lock to the Rails code base, and made the necessary changes to the Rails default Mongrel script so that this handler would get loaded by Mongrel in lieu of the one packaged with Mongrel.

actionpack/lib/action_controller/dispatcher.rb

```
def dispatch
```

```
  run_callbacks :before  
    handle_request  
  rescue Exception => exception  
    failsafe_rescue exception  
  ensure  
    run_callbacks :after, :reverse_each
```

```
  @@guard.synchronize do
```

```
    begin  
      run_callbacks :before  
      handle_request  
    rescue Exception => exception  
      failsafe_rescue exception  
    ensure  
      run_callbacks :after, :reverse_each  
    end
```

```
  end
```

```
end
```

<http://github.com/rails/rails/commit/e2d4ebdea4eab41c4af1c5530a9e180d11529dec>

...And in the same change set, he added the lock to the Dispatcher code inside of Rails. That way it would be easier for Rails committers to adjust it as the Rails code base becomes more thread-safe.

UPDATE: Since I gave this talk, Rails 2.1 has been released. This is the Rails 2.1 Dispatcher#dispatch. **<=**

actionpack/lib/action_controller/dispatcher.rb

```
def dispatch
```

```
  def handle_request
```

```
    @controller = Routing::Routes.recognize(@request)
```

```
    @controller.process(@request, @response).out(@output)
```

```
  end
```

```
  run_callbacks :after, :reverse_each
```

```
  @@guard.synchronize do
```

```
    begin
```

```
      run_callbacks :before
```

```
      handle_request
```

```
      rescue Exception => exception
```

```
        failsafe_rescue exception
```

```
      ensure
```

```
        run_callbacks :after, :reverse_each
```

```
      end
```

```
    end
```

```
  end
```

<http://github.com/rails/rails/commit/e2d4ebdea4eab41c4af1c5530a9e180d11529dec>

Here's another view of the Mutex-wrapped Dispatcher code with arrows pointing to the methods protected by the mutex, and the source for the protected method, handle_request.

What Ezra did in dispatch_rack, which currently only exists in his branch, is split out the Routing::Routes.recognize call and the controller.process call from handle_request...

```
def dispatch_rack(env = {}, session_options =
  ActionController::RackRequest::DEFAULT_SESSION_OPTIONS)
  @request = RackRequest.new(env, session_options)
  @response = RackResponse.new
  begin
    run_callbacks :before_dispatch
    @controller = Base.router.recognize(@request)
    @@guard.synchronize do
      @controller.process(@request, @response)
    end
    rescue Exception => exception
      failsafe_rescue exception
    ensure
      run_callbacks :after_dispatch,
      :enumerator => :reverse_each
    end
    rescue Exception => exception
      failsafe_rescue exception
    end
  end
```



...so that he could see what would happen if they were called outside the lock.

He hasn't run it very much yet, but so far the results are promising. I want to emphasize though, that despite the fact that testing can be helpful, it does not mean that something is thread-safe just because it was threaded and nothing bad happened.

```

module ActionView
  class TemplateFinder
    def process_view_paths(*view_paths)
      view_paths.flatten.compact.each do |dir|
        next if @@processed_view_paths.has_key?(dir)
        @@processed_view_paths[dir] = []
        (Dir.glob("#{dir}/**/*/**")
         | Dir.glob("#{dir}/**")).each do |file|
          unless File.directory?(file)
            @@processed_view_paths[dir] << file.split(dir).last.sub(/^\/$/, '')
            extension = file.split(".").last
            if template_handler_extensions.include?(extension)
              key = file.split(dir).last.sub(/^\/$/, '').sub(/\.(\w+)$/ , '')
              @@file_extension_cache[dir][key] << extension
            end
          end
        end
      end
    end
  end
end

module ActionView
  module TemplateHandlers
    module Compilable
      def create_template_source(template, render_symbol)
        body = compile(template)
        self.template_args[render_symbol] ||= {}
        locals_keys = self.template_args[render_symbol].keys
        | template.locals.keys
        self.template_args[render_symbol] =
          locals_keys.inject({}) { |h, k| h[k] = true; h }

        ...
        "def #{render_symbol}(local_assigns)\n#{locals_code}#{body}\nend"
      end
    end
  end
end

```

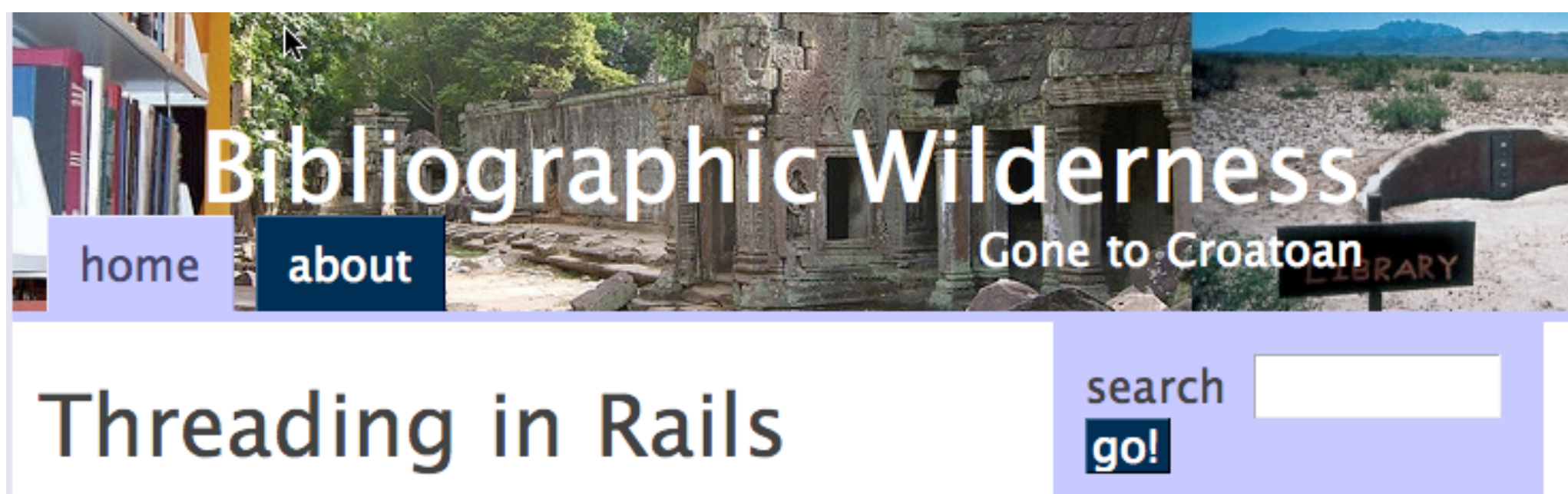
In both methods shown here, the code that modifies class variables is not thread-safe. There’s not much to look at. What you can’t see here is that these methods represent the kind of refactoring I’ve seen in a number of places since Rails 2.0 was released. Both the Compilable module and the TemplateFinder are constructed from methods that were loose in the ActionView file. There are now fewer class variables in ActionView and it’s easier to identify where synchronization is needed than when these class variable were scattered throughout ActionView.

UPDATE: The TemplateFinder code has been reconstituted using the new ActionView::ViewLoadPaths class in Edge Rails. <=

There are few additional known thread-safety issues in ActionController and ActiveSupport. In the course of his analysis Josh will likely find some others.

Now let’s move on to ActiveRecord.

For some reason the question of whether ActiveRecord is thread-safe seems have an air of mystery around it.



I decided to buck the community conventional wisdom and try to use [Ruby Threads](#) in a Rails app.

Jonathan Rochkind

<http://bibwild.wordpress.com/2007/08/28/threading-in-rails/#more-53>

I know of some people who have used and are continuing to use ActiveRecord in threaded mode without incident.

Jonathan Rochkind wrote about his experiences with using ActiveRecord with threads in his blog last summer. I touched base with him a few weeks ago, and he confirmed that the application he wrote about is still in production, and he had not had any problems with it.

He needed to query Google, Yahoo, and some other search engine APIs, so he started a thread for each Web service. And wanted to process all the results before returning a response to the user. He accomplished this by calling join on each thread, also from the same controller action.

He explains that he was careful to give each thread its own copy of any Rails framework classes it needed to do its work.

One other thing I'd like to point out about this example, is that it's a great example of where userspace threads can be very useful. As I mentioned before, two userspace threads can't actually run at the same time, they only appear to run simultaneously because the VM time slices between them so quickly.

But there's a clear benefit to getting the queries started for each search engine as close to the same time as possible, as opposed to querying one and waiting for the results to come back before initiating a query with the next one.

In his blog entry, Jonathan also describes his home-grown solution for handling long-running tasks. He starts the jobs in threads in a controller action, but does not call "join" for any threads. As they progress, the jobs deposit status values in the database, where they can be retrieved on demand or periodically via AJAX, and displayed to the user.

I know, and Jonathan acknowledges in his blog, that the recommended way to handle asynchronous processing is to use a plugin that creates background jobs. I don't want to focus on that aspect of his design in this talk, but here's a link to the slides for a RailsConf 2008 talk about background processing: <http://railspikes.com/2008/6/3/asynchronous-railsconf-2008>

Not many people are running active record in threaded mode, and of those who are, fewer have been successful with it. Jonathan is in the minority.

I've read a number of mailing list posts that that say things like "When I try multi-threading with ActiveRecord ...

Weird Things Happened

.... weird things happen!”

And yet there are no known ActiveRecord bugs. No one has reported these bugs in the bug tracker.

I think there are several reasons for the confusion and uncertainty. One is that thread-safety bugs are not always easy to reproduce. But there are also reasons that are particular to ActiveRecord.

First of all, Rails opens one connection per thread, and doesn’t automatically close them when the thread dies. So if you create 100 threads, Rails will try to open 100 database connections. It’s easy to reach the maximum number of allowed connections if you don’t implement your own connection pool or otherwise manually manage database connection allotment.

There’s actually a very simple way to handle the proliferation of connections, that may be all the connection management that’s needed for basic threading scenarios, such as the ones Jonathan Rochkind blogged about. Calling `verify_active_connections` takes care of closing the connections for any threads that are no longer active -- but it’s never been very well publicized. It can be called after threads are finished with their work.

Also, as we saw ActionPack is not thread-safe. Some of the weird problems developers may have had can probably be attributed to developers spinning threads from within Rails controllers and running into weird ActionPack thread-safety bugs.

There’s the possibility that they ran into ActiveRecord bugs that have nothing to do with thread-safety. It has not seen a lot of use in that mode; it has not been put through its paces in many multi-threaded environments.

```
environment.rb
```

```
ActiveRecord::Base.allow_concurrency = true
```

Another reason why people might have had trouble with ActiveRecord is that apparently it is not common knowledge that you need to set `ActiveRecord::Base.allow_concurrency` to `true` in `environment.rb` in order to run in threaded mode. Otherwise, Rails will only allow a single database connection.

It's false by default. I don't think the problem is that developers expect it to be true by default. From mailing list archives it looks like a fair number of developers just didn't know the flag existed.

```
module ActiveRecord
  class Base
    class ConnectionSpecification
      class << self

        def thread_safe_active_connections
          @@active_connections[Thread.current.object_id] ||= {}
        end

        def single_threaded_active_connections
          @@active_connections
        end

        if @@allow_concurrency
          alias_method :active_connections,
                      :thread_safe_active_connections
        else
          alias_method :active_connections,
                      :single_threaded_active_connections
        end

        ...
      end
    end
  end
end
```

The code fragment here shows where Rails looks at the ActiveRecord's `allow_concurrency` flag, and how ActiveRecord stores the connections by `Thread.current.object_id` in an `@@active_connections` hash, if `allow_concurrency` is set to true.

ActionController::Base

Rails 2.0.2

```
# Controls whether the application is thread-  
# safe, so multi-threaded servers like WEBrick  
# know whether to apply a mutex  
# around the performance of each action.  
# Action Pack and Active Record are by default  
# thread-safe, but many applications  
# may not be. Turned off by default.
```

```
@@allow_concurrency = false  
cattr_accessor :allow_concurrency
```

Rails 2.1

```
# Indicates to Mongrel or Webrick whether to  
# allow concurrent action processing. Your  
# controller actions and any other code they call  
# must also behave well when called from  
# concurrent threads. Turned off by default.
```

It probably doesn't help matters that there's an @@allow_concurrency flag in ActionController::Base, as well as an @@allow_concurrency flag in ActiveRecord::Base.

The ActionController::Base comment on the top of this slide is confusing and misleading. It says that ActionPack and ActiveRecord are thread-safe by default, which could lead developers to think it's okay to run threaded code within Rails without protecting the framework classes.

I don't believe that this comment was meant to be misleading. I just think no one on the Rails core team had looked at it in a long time.

As soon as I pointed this comment out to Jeremy Kemper when I noticed it as I was preparing this talk, he changed it to the clear comment you see on the bottom part of the slide.




UPDATE: the @@allow_concurrency flag in ActionController::Base has been removed in Edge Rails. <=

ActiveRecord::Base.allow_concurrency = true



Still there's one unexplained mystery. Ezra claims that once when he was experimenting with ActiveRecord with `allow_concurrency` set to `true` he saw the results for a query that was executed in one thread appear in a different thread. Since he's an expert in the field, we can rule out some of the reasons I just mentioned as potential explanations for what he saw. He's well aware of connection management semantics and knows the implications of the various settings.

I touched base with him and asked him if he had seen this behavior recently. He said that he has not run AR in threaded mode since having that experience. He said that "the per thread connections and manual cleanup" are part of why he recommends using a lock around any Merb code that references ActiveRecord, but that the main reason why he did not have a lot of incentive to spend time tracking down the source of what he characterized as hard-to-reproduce bugs was that he found that with green threads, ActiveRecord performed much better in single-threaded mode than in locked\threaded mode.

nicksieger / rails   

Fork of rails/rails

Description: Ruby on Rails

Homepage: <http://rubyonrails.org>

Clone URL: <git://github.com/nicksieger/rails.git>

Repo Search:

all branches **all tags**

- 1-2-stable
- 2-0-stable
- connection_pool
- databases_rake_regexp
- master

`activerecord/lib/active_record/connection_adapters/abstract/connection_pool.rb`

```
def allow_concurrency=(flag)
  if @@allow_concurrency != flag
    @@allow_concurrency = flag
    if flag
      self.connection_pools_lock = Monitor.new
    else
      self.connection_pools_lock = NullMonitor.new
    end
  end
end
```

That mystery may never be solved, but in the future there will not be so much mystery around thread safety and ActiveRecord.

For one thing Josh will be including ActiveRecord in his analysis and putting it through its paces along with the rest of the Rails code base.

For another, JRuby team member Nick Sieger is working on adding connection pool support. When he's ready to check in his code, the plan is for the connection pool to be used regardless of whether you are running multiple threads. The same logic executes even if the pool size is 1. Manual connection management will no longer be required.

The old ConnectionSpecification file had a fair amount of branching logic, with different paths taken depending on whether allow_concurrency was true or not.

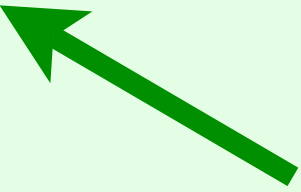
The code Nick has added to AR ConnectionSpecification is very DRY.

The only place the allow_concurrency flag needs to be checked is where the pool determines whether to use a null mutex or a real one. (Actually he's using a monitor here, not a mutex. The main difference between a monitor and a mutex is that you can have nested locking statements with a Monitor.)

activerecord/lib/active_record/connection_adapters/postgresql_adapter.rb

```
@async = config[:allow_concurrency]
```

```
def execute(sql, name = nil) #:nodoc:
  log(sql, name) { @connection.exec(sql) }
  log(sql, name) do
    if @async
      @connection.async_exec(sql)
    else
      @connection.exec(sql)
    end
  end
end
```



Here's another patch that, like connection pool support, adds something that makes ActiveRecord more friendly to threading, though not more thread-safe.

It calls the async version of exec, instead its synchronous counterpart, on the native postgres driver.

Without this patch, Rails was basically unusable with the native postgres driver in multithreaded mode. When one userspace thread is blocked, all the other threads in the same process are blocked. So any time one thread had to wait for a db resource, none of the other threads could service requests.

Now the native postgres driver async_exec implementation tests the connection to see if the server is busy before invoking the db. If the server is busy, the thread is put to sleep until the server is available -- giving the Thread Scheduler an opportunity to context-switch to other threads.

A similar patch was added for the Oracle native drivers.

Actually, any C extension would cause similar blocking issues with userspace threads without some kind of async support.

Now that we've seen some of the reasons why Rails is not thread-safe, we'll take a high-level look at different ways that developer working with different implementations of Ruby have addressed the memory, ease of use and speed issues that have plague typical processed-based deployments.

We'll start with Ruby 1.8.

Evented Mongrel



Ebb

One option Rails developers have is Single Threaded event-driven deployments using Evented Mongrel, Thin or Ebb. Evented Mongrel has been around for a little more than a year, but both Thin and Ebb are relatively new. All three servers support both threaded and event-driven modes.

In event-driven mode, they only process one request at a time by design, so it seems counter-intuitive that they could perform better in event-driven mode than in threaded mode -- but the overhead from running threads and context switching should not be discounted. And with Ruby 1.8, there's no way to leverage multiple cores. So there's overhead without a lot of payoff. It's not uncommon for Merb developers to use these servers in evented mode even though they have the option of running in threaded mode.

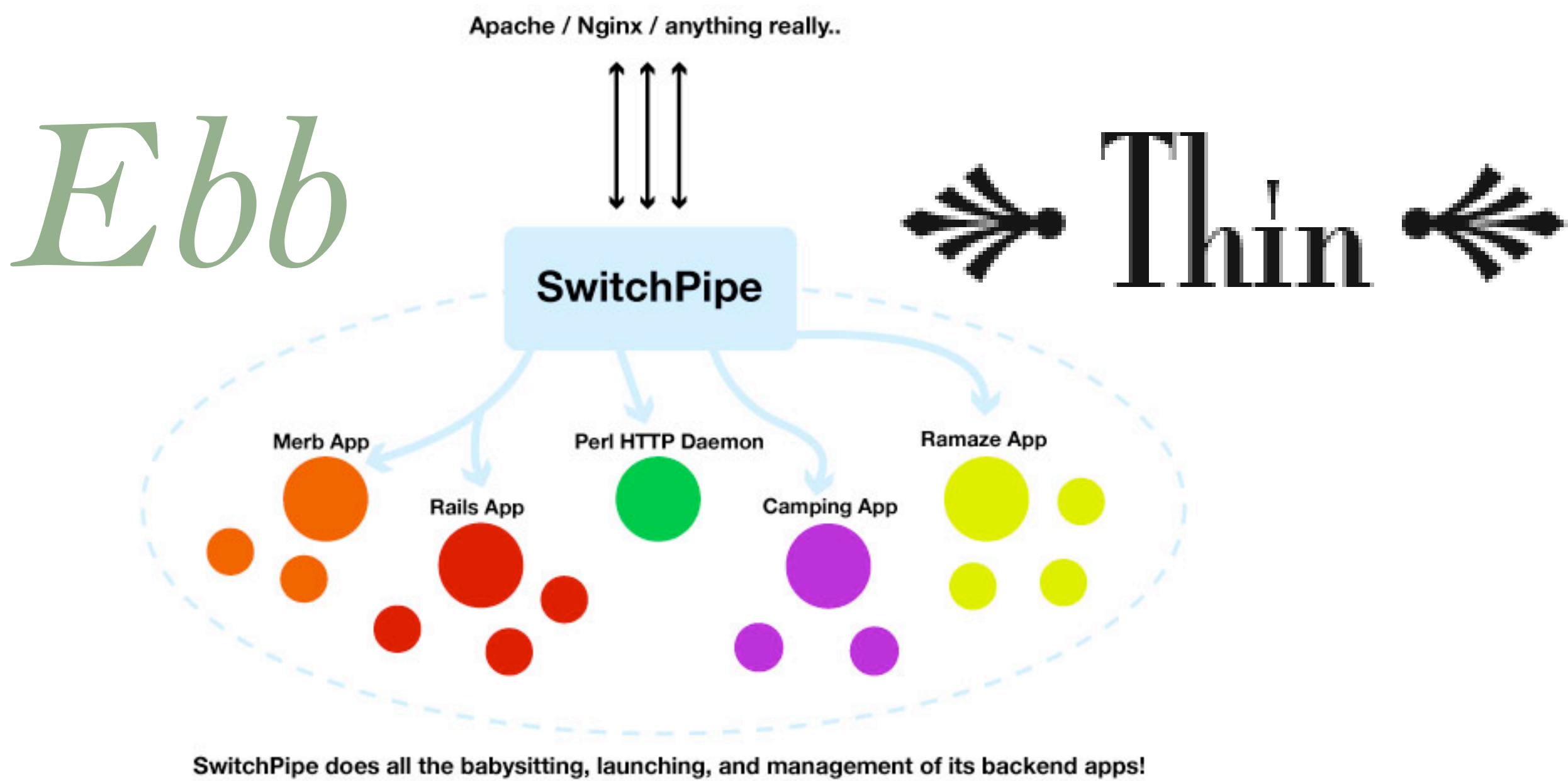
Where evented mode doesn't work so well is for slow actions, which bring the event loop to a standstill.

Both Thin and Ebb now offer a solution, but it won't work for Rails as long as Rails isn't threadsafe. You can list long-running actions in a config file, and there's a mechanism in place for them to spawn threads for long-running actions so they won't block the rest of the event loop.

Some links:

- Evented Mongrel (<http://swiftply.swiftcore.org/mongrel.html>)
- Thin (<http://code.macournoyer.com/thin/>)
- Ebb (<http://ebb.rubyforge.org/>)

SwitchPipe



SwitchPipe (<http://switchpipe.org/>) represents an improvement over standard clustering.

Its logo characterizes it as doing all the “babysitting, launching, and management of its backend processes.”

It sits in front of a Web Server like Apache or Nginx and launches, and then manages the number of apps its configured to handle based on a supplied min and max number of processes and a time out.

SwitchPipe does what it can on the memory management side. It provides over a typical cluster is that it will shut down instances based on a specified time-out, so the max number of instances are not always running.

It emphasizes ease of use over performance, but setting the min number of processes to 1, yields modest performance gains. If one process is kept running, there’s no start-up penalty.



Litespeed (<http://www.litespeedtech.com/>) is not open source, but a free edition is available in addition to the enterprise edition. It spawns Rails applications on demand, manages them, and takes care of load balancing.

Speed is the attribute that it emphasizes the most in its literature. A big part of their approach to reducing startup time is to load the Rails framework and then fork child processes. I explained earlier that a child process starts off life as a replica of the parent process. Leveraging fork, Litespeed only incurs the overhead associated with loading the framework once for all the spawned Rails application instances.

Additional speed gains come from Litespeed's optimized communication protocol.

I should add that there's been some confusion about whether multiple cores are ever utilized with the free edition -- partly because there's a setting called multi-core that is set to 1 in its configuration. As it turns out there's nothing to stop the application instances from being routed to multiple cores with the free edition. The multi-core setting just controls whether the lsshttpd processes are mapped to multiple cores. Serving static assets is an example of what might be impacted by the setting.



Passenger (<http://www.phusion.nl/products.html>) is the newest server on the block.

It makes deployment on shared hosts as easy as just uploading files. Built on top of Apache, Passenger fills a void in the shared host space, where Ruby applications frequently require special handling.

The Passenger team has also focused on performance, posting impressive numbers.

However, what makes Passenger stand out more than anything else, in my opinion, is the way it addresses the memory issue.

Memory



<http://www.flickr.com/photos/56871332@N00/197941804/sizes/m/>

<http://www.flickr.com/photos/56871332@N00/197941804/sizes/m/>

As you have seen there has been a lot done to improve on the cluster-based model for deploying Rails apps in the area of user-friendliness, including both initial deployment and maintenance. Performance has been addressed as well.

But none of the other options we looked at so far make a significant dent in the memory problem.

Memory



<http://www.flickr.com/photos/56871332@N00/197941804/sizes/m/>



<http://www.flickr.com/photos/56871332@N00/197941804/sizes/m/>

The Phusion developers who created Passenger, found an innovative way to make a huge difference.

They knew that forking subprocesses after loading Rails could yield much greater memory savings than it did in practice. Unix systems support a policy called "copy on write", which allow a child process and a parent process to share the same memory up until the point where one of them makes a change. they had seen it work with Perl, but Ruby's garbage collector prevents this policy from providing any memory savings because of the way it marks objects themselves, as opposed to keeping track of object status in a table. Once Ruby's garbage collector marks a page, the OS will assume it can't be shared -- even if the child process didn't make any changes to it.

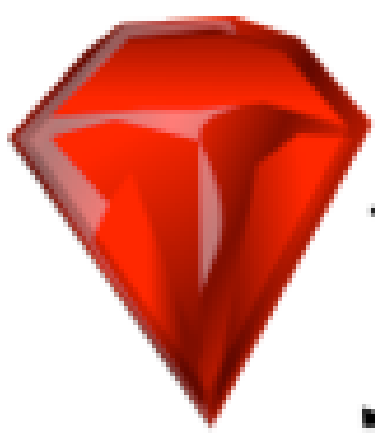
Phusion created a patch that makes Ruby "copy-on-write" friendly. They are offering it to the Ruby community for free. To find out the latest about whether the patch will make it into Ruby 1.9 -- and to learn more details about Passenger, you'll have to go to the Passenger talk on Sunday, June 1.

UPDATE: Here are the slides from the Passenger talk at RailsConf on June 1: http://assets.en.oreilly.com/1/event/6/mod_rails_%20Easy%20and%20Robust%20Deployment%20of%20Your%20Ruby%20on%20Rails%20Applications%20on%20Apache%20Webservers%20Presentation.pdf

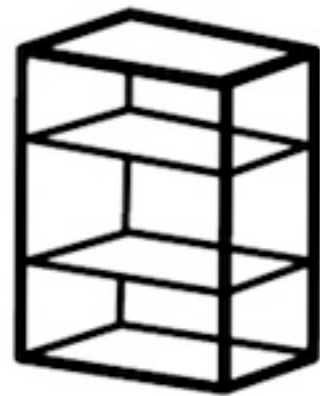
Since June 1, here is how the Phusion Ruby Enterprise Edition site FAQs (<http://www.rubyenterprisedition.com/faq.html#fork>) address the question "Why did you fork Ruby?":



<



JRuby



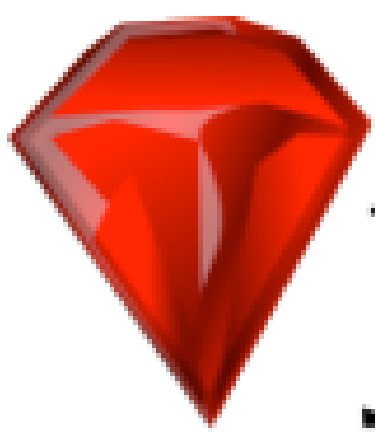
rack
powers web applications

```
public void process
(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    final RackApplicationFactory rackFactory = getRackFactory
    ();
    RackApplication app = null;
    try {
        app = rackFactory.getApplication();
        app.call(request).respond(response);
    } catch (Exception re) {
        handleException(re, rackFactory, request, response);
    } finally {
        if (app != null) {
            rackFactory.finishedWithApplication(app);
        }
    }
}
```

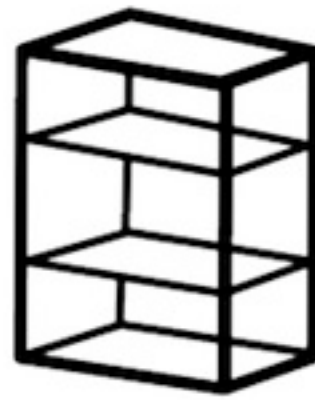
Multiple JRuby VMs can run in a single process. This is important feature for JRuby, not just because its support for deploying Rails is based on it, but also for the way it implements system calls. Without the MVM mechanism, JRuby would need to start a new process for every system call, and starting a new process is expensive for JRuby, since it means spawning a new Java Virtual Machine.

The JRuby strategy for handling multiple concurrent Rails requests, given that Rails is not threadsafe, is to use a pool of JRuby VMs, each with the Rails environment pre-loaded. Needing to run multiple VMs per Rails application may sound like an expensive proposition memory-wise, but it helps that the VMs can share JVM bytecode, while still providing application-level isolation.

With the new JRuby-Rack-based deployment mechanism (<http://git.caldersphere.net/?p=jruby-rack.git;a=summary>), the dispatcher, shown here is the same for a Rails installation as it is for Merb and other thread-safe frameworks, but ...



JRuby



rack
powers web applications

```
public void process
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
final RackApplicationFactory rackFactory = getRackFactory();
RackApplication app = null;
try {
    app = rackFactory.getApplication();
    app.call(request).respond(response);
} catch (Exception e) {
    handleException(e);
} finally {
    if (app != null)
        rackFactory.release(app);
}
}
```



```
public class PoolingRackApplicationFactory
implements RackApplicationFactory {
    ...
    public RackApplication getApplication()
        throws RackInitializationException {
        return realFactory.getApplication();
    }
    ...
}
```

..Rails deployments provide a PoolingRackApplicationFactory where deployments that can share a single runtime supply a non-pooling factory.

As far as ease of deployment is concerned -- a one-liner with a tool called Warbler generates a .war file based on your Rails app. It can be dropped into an app server.

Here's JRuby's home page: <http://jruby.codehaus.org/>

mod_rubinius

The Rubinius team's mod_rubinius is not a Rails-centric solution, but it sounds like it's going to be a boon to Rails developers none-the-less.

As mod_rubinius lead developer Eero Saynatkari (a.k.a rue) explained it to me, mod_rubinius is really going to be ...

mod_Rubinius

...Rubinius, itself. The "mod" part will involve little more than building some lightweight adapters into Rubinius to that it can easily interface with applications and Web servers.

Rubinius will feature serverization capabilities, including process management, dispatching and load balancing services. It will be possible to run it in cluster mode, specifying a number of processes per cluster and a number of VMs per process.

Some of the reasons why using mod_rubinius to deploy applications can reduce memory requirements because rubinius has advanced copy-on-write semantics, and the VMs in a cluster can share bytecode.

Future plans for mod_rubinius include support for distributed deployments. Eero suggested that once an infrastructure is in place for that -- Rubinius should be able to offer an object space sharable by VMs on physically different locations.

For more information about Rubinius see the Rubinius home page (<http://rubini.us/>) and chat logs (<http://donttreadonme.co.uk/rubinius-irc/>).

Rubinius

+



```
pool = (0...POOL_SIZE).map {  
  Rubinius::VM.spawn "rbx_mongrel.rb"  
}  
loop do  
  
  # server is a regular TCPServer  
  socket = server.accept  
  vm = pool.shift  
  
  # send the file descriptor num to the other vm  
  vm << socket.fileno  
  pool.push vm  
end
```

mod_rubinius development just got started in earnest so I can't show you mod_rubinius code that can give you a less abstract idea of what VM-based deployment might look like code-wise. Rubinius committers Tony Arcieri and MenTaLguY did however provide me with a snippet of code that shows how to run instances of Mongrel, which could each be running Rails, in multiple VMs. It routes requests to the VM pool in round robin fashion.

This code starts Mongrel running in multiple VMs, by virtue of passing rbx_mongrel.rb, a modified version of the Mongrel start-up script to VM.spawn. It then loops through them, passing a client socket file descriptor to a VM in each iteration of the loop. The socket file descriptor needs to be passed instead of the IO object because the multiple VM API can only dispatch primitive types at this time.

Rubinius

+



```
pool = (0...POOL_SIZE).map {  
  Rubinius::VM.spawn "rbx_mongrel.rb"  
}  
loop do
```

```
  # server i  
  socket = s  
  vm = pool.  
  
  # send the  
  vm << sock  
  pool.push  
end
```

```
def run  
  while true  
    # client = @socket.accept  
    fd = Rubinius::VM.get_message  
    # create a local IO object for this fd  
    client = IO.for_fd(fd)  
    thread = Thread.new(client)  
    { |c| process_client(c) }  
    thread[:started_on] = Time.now  
    sleep @throttle if @throttle > 0  
    graceful_shutdown  
  end  
end
```

Here is an abridged version of the modified Mongrel start-up routine from `rbx_mongrel.rb`. You can see where the standard code for accepting the client socket is commented out, and a VM is receiving the socket file descriptor.

IronRuby

```
public static void Main() {
    const string write = @"C:\Temp\write.rb";
    const string read = @"C:\Temp\read.rb";

    ...
    # code that writes Ruby code to 'write.rb'

    ...
    # code that writes Ruby code to 'read.rb'

    ScriptRuntime runtime1 = ScriptRuntime.Create();
    ScriptRuntime runtime2 = ScriptRuntime.Create();

    runtime1.ExecuteFile(write);
    runtime2.ExecuteFile(read);
    runtime1.ExecuteFile(read);
}
```

It was only this week that IronRuby ran Rails framework code for the first time, but the IronRuby team has long had a plan for optimal Rails deployment.

The IronRuby answer for a framework that that's not thread-safe is that its possible to run multiple isolated IronRuby instances in the same CLR AppDomain.


Because Rails is not yet running in that configuration, I can't show you sample code from an actual Rails installation, but Tomas Matousek of the IronRuby team provided this sample code that demonstrates the extent to which IronRuby runtimes in the same AppDomain can be isolated from one another.

IronRuby

```
public static void Main() {
    const string write = @"C:\Temp\write.rb";
    const string read = @"C:\Temp\read.rb";

    ...
    # code that writes Ruby code to 'write.rb'
    ...
    # code that wFile.WriteAllText(write, @"
        $x = 'Hello from runtime #1!'
        C = 'some constant'
        module Kernel
            def say_bye
                puts 'bye'
            end
        end
    " );
}

ScriptRuntime
ScriptRuntime
runtime1.Exec
runtime2.Exec
runtime1.Exec
```



```
File.WriteAllText(write, @"
    $x = 'Hello from runtime #1!'
    C = 'some constant'
    module Kernel
        def say_bye
            puts 'bye'
        end
    end
");
```

First this Ruby code that defines a constant and a “say_bye” method is written to a Ruby file called write.rb.


IronRuby

```
public static void Main() {
    const string write = @"C:\Temp\write.rb";
    const string read = @"C:\Temp\read.rb";

    ...
    # code that writes Ruby code to 'write.rb'

    # code that writes Ruby code to 'read.rb'

    ScriptRuntime
    ScriptRuntime
    runtime1.Exec
    runtime2.Exec
    runtime1.Exec
}
```



```
File.WriteAllText(read, @"
    puts $x
    if defined? C
        puts C
    else
        puts 'C not defined'
    end
    say_bye rescue puts $!
    puts
");
```

Next some Ruby code is written to “write.rb”. It will print out the constant if it is defined. The code here also calls the say_bye method.

IronRuby

```
public static void Main() {
    const string write = @"C:\Temp\write.rb";
    const string read = @"C:\Temp\read.rb";

    ...
    # code that writes Ruby code to 'write.rb'

    ...
    # code that writes Ruby code to 'read.rb'

    ScriptRuntime runtime1 = ScriptRuntime.Create();
    ScriptRuntime runtime2 = ScriptRuntime.Create();

    runtime1.ExecuteFile(write);
    runtime2.ExecuteFile(read);
    runtime1.ExecuteFile(read);
}
```

Next two IronRuby Runtimes are created in the same AppDomain. You can pass an AppDomain to a Runtime, but if you pass nothing, the current AppDomain is used.

IronRuby

```
public static void Main() {
    const string write = @"C:\Temp\write.rb";
    const string read = @"C:\Temp\read.rb";

    ...
    # code that writes Ruby code to 'write.rb'

    ...
    # code that writes Ruby code to 'read.rb'

    ScriptRuntime runtime1 = ScriptRuntime.Create(domain);
    ScriptRuntime runtime2 = ScriptRuntime.Create(domain);

    runtime1.ExecuteFile(write);
    runtime2.ExecuteFile(read);
    runtime1.ExecuteFile(read);
}
```

Only runtime1 executes the code that defines the method and the constant. Then both runtimes execute the code that tries to read the constant and call the method.

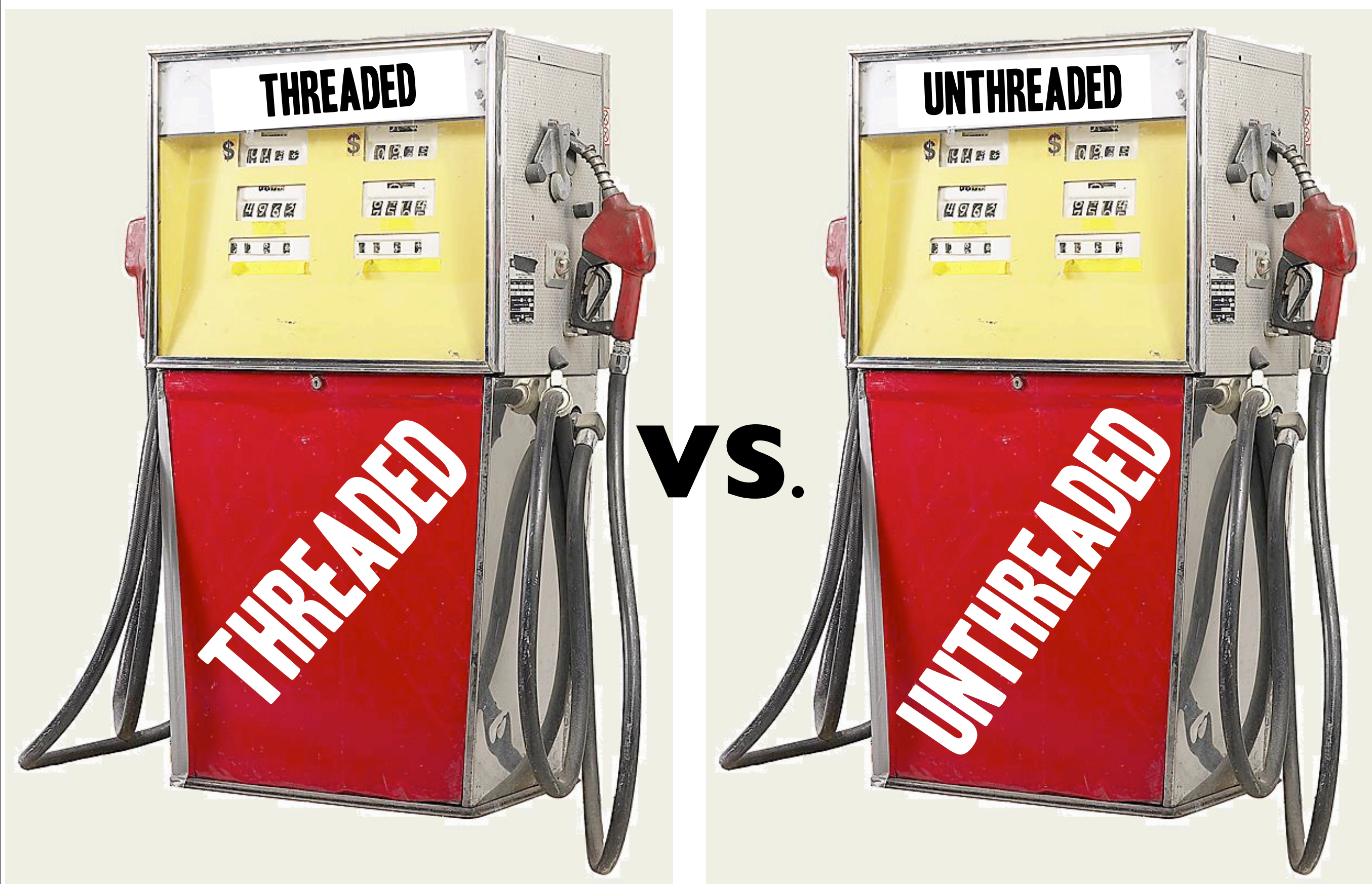
IronRuby

```
public static void Main() {
    const string write = "C:\\IronRuby\\Bin\\Debug>rt.exe";
    const string read = "C:\\IronRuby\\Bin\\Debug>";
    const string nil = "C not defined";
    ...
    # code that will write to the console
    ...
    # code that will read from the console
    AppDomain domain = AppDomain.CreateDomain("IronRuby", null, null);
    ScriptRuntime runtime1 = ScriptRuntime.Create(domain);
    ScriptRuntime runtime2 = ScriptRuntime.Create(domain);

    runtime1.ExecuteFile(write);
    runtime2.ExecuteFile(read);
    runtime1.ExecuteFile(read);
}
```

```
C:\IronRuby\Bin\Debug>rt.exe
C not defined
undefined local variable or method
`say_bye' for main:Object
Hello from runtime #1!
some constant
bye
```

Here's the output. You can see that the method and the constant were only defined for runtime1, even though they are in the same AppDomain.



http://governing.typepad.com/photos/uncategorized/gas_pump.jpg

We just looked at ways the the deployment picture for single-threaded Rails is improving. At the same time it looks like it may become safe to run Rails in multi-threaded mode in the not-too-distant future.

This does not mean that the single-threaded options are going to be abandoned, it just means that developers have more choices.



```
use_mutex: false
-- mutex off
```

```
def dispatch_action(klass, action, request, status=200)
  # build controller
  controller = klass.new(request, status)
  if use_mutex
    @@mutex.synchronize {
      controller._dispatch(action)
    }
  else
    controller._dispatch(action)
  end
  controller
end
```

The Merb framework is thread-safe, but supports a “use mutex” option that you can set on the command line or in merb.yml to indicate whether or not you want a Mutex lock around your application-level code.

Rails will likely support a similar option. In order to safely run on a threaded server without a mutex, you’ll need to ensure that you application level code is thread-safe and that any libraries or plugins you use are thread-safe.

While there’s no substitute for analyzing code for thread-safety violations, you might want to look ...

Mongrel Debug



```
class Threads < GemPlugin::Plugin "/handlers"
  include Mongrel::HttpHandlerPlugin
  def process(request, response)
    MongrelDbg::trace(:threads,
      "#{Time.now} REQUEST #{request.params['PATH_INFO']}")
    ...
    ObjectSpace.each_object do |obj|
      ...
      if obj.class == Mongrel::HttpServer
        worker_list = obj.workers.list
        if worker_list.length > 0
          keys = "-----\n\tKEYS:"
          worker_list.each {
            |t| keys << "\n\t\t-- #{t}: #{t.keys.inspect}" }
          end
          MongrelDbg::trace(:threads,
            "#{obj.host}:#{obj.port}--THREADS: #{worker_list.length} #{keys}")
        end
      end
    end
  end
end
```

...at Mongrel's built-in debugger, which can be configured to log files that files that are open between processes you run Mongrel with the -B option it will run this thread-debugging code which prints out a list of any threads that are still alive between processing requests.

Workers is a ThreadGroup that is set up in the main Mongrel event loop. Once a threadgroup is set up, any new threads created during the servicing of a request will automatically be added to it. Calling list on a threadgroup gives you a list of any threads that have not terminated.

the { buckblogs :here }

Net::SSH and Thread-safety

Posted by Jamis on March 17, 2008 @ 09:55 PM

[I]n the interest of getting feedback from people who might actually use the library, I ask you: which would you prefer? A faster library in single-threaded programs? Or a simple program in multi-threaded ones? Is there a general best-practice in this case?

Jamis Buck

<http://weblog.jamisbuck.org/2008/3/18/net-ssh-and-thread-safety>

Net::SSH is an example of a library that not thread-safe. If a library is not thread-safe it doesn't necessarily mean its author was careless.

While he was working on a new Net::SSH library, Former Rails Core Team member Jamis Buck wrote a blog post asking whether developers would prefer for him to make the library thread-safe or not.

The discussion on his blog was interesting. The suggestion was made to add a configurable mutex.

He decided to go with not making the library thread-safe. Users of the library need to handle their own synchronization.

I think we're going to be seeing the emergence of more libraries that wrap the concurrency primitives. We'll be able to count on them to get the synchronization details right, not unlike the way it's standard practice to rely on JavaScript libraries to handle the cross-browser particulars and low-level details.

According to Eero, some of the Rubinius enhancements required for cluster mode support will constitute a concurrency library of sorts. Rubinius will be able to provide transparent parallelization services for non-server applications.

Just in the last couple of months two small concurrency libraries were introduced....

NAME

forkoff

SYNOPSIS

brain-dead simple parallel processing for ruby

URI

<http://rubyforge.org/projects/codeforpeople>

INSTALL

gem install forkoff



Parallel Each (for ruby with threads)

In the simplest case, you are one letter away from harnessing the power of parallelism

<http://peach.rubyforge.org/?peach>

...forkoff, which claims “brain-dead simple parallel processing for Ruby” and peach, with a home page that asserts “In the simplest case, you are one letter away from harnessing the power of parallelism.”

If you use “peach” instead of “each” as your iterator, the logic in the block you pass to peach will be executed in a separate thread for each item in an Array. You can also specify a number of threads .

As it’s name suggests, forkoff’s implementation is process-based. The default number of subprocesses it spawns is two, but its configurable.

NAME

forkoff

```
s=[5,2,3,4].forkoff{|n| 10+n}
```

SYNOPSIS

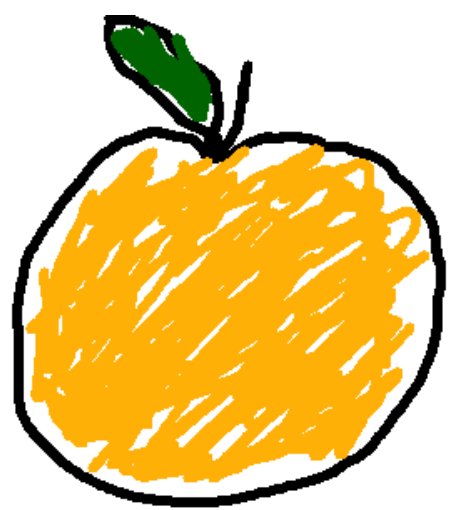
brain-dead simple parallel processing for ruby

URI

<http://rubyforge.org/projects/codeforpeople>

INSTALL

```
gem install forkoff
```



Peach

```
[1,2,3,4].peach{|n| puts n}
```

In the simplest case, you are one letter away from harnessing the power of parallelism

<http://peach.rubyforge.org/?peach>

Here are examples of the basic usage.

```

class Array
  def peach(n = nil, &b)
    peachrun(:each, b, n)
  end
  def pmap(n = nil, &b)
    peachrun(:map, b, n)
  end
  def pdelete_if(n = nil, &b)
    peachrun(:delete_if, b, n)
  end
  protected
  def peachrun(meth, b, n = nil)
    ...
  end

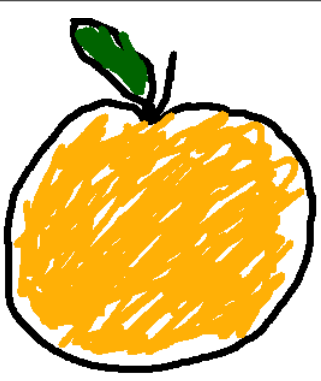
  def divvy(n = nil)
    n ||= $peach_default_threads || size
    ...
  end
end
end

```



Peach supports pmap and pdelete_if in addition to peach. It adds these methods to Ruby's Array class.

All three methods call peachrun behind the scenes, to spin the threads.



Peach

[1,2,3,4,5,6,7,8,9]

```
def peachrun(meth, b, n = nil)
  threads, results, result = [], [], []
  divvy(n).each_with_index do |x,i|
    if x.size > 0
      threads << Thread.new {
        results[i] = x.send(meth, &b)}
    else
      results[i] = []
    end
  end
  threads.each {|t| t.join }
  results.each {|x| result += x}
  result
end
```

[[1,2,3][3,4,6][7,8,9]]

<http://peach.rubyforge.org/?peach>

The divvy method is what breaks down the array into smaller batches to be processed in separate threads. So, if you pass a 9-member array to peach, with the “thread number” set at 3, the data will be processed in 3 concurrent batches.

Forkoff: Producers & Consumers w/SizedQueue

Producers

```
producers = []  
  
n.times do |i|  
  thread = Thread.new do  
  
    ...  
    each_with_index do |args, j|  
      every_nth = j.modulo(n) == i  
      next unless every_nth  
      qs[ j.modulo(n) ].push( [args, j] )  
    end  
    qs[ i ].push( Forkoff.done )  
  end  
  
  producers << thread  
end
```


Consumers

```
consumers = []  
  
n.times do |i|  
  thread = Thread.new do  
  
    ...  
    pid = fork  
    ...  
  end  
  
  consumers << thread  
end
```

Forkoff on the other hand does not do its processing in batches. It spawns the specified number of processes using fork, starts each process off with initial data, and then feeds data to processes as they become free. Forkoff employs the producer \consumer abstraction. Although it uses fork to create processes, but it uses threads to manage the processes.

Forkoff: **Producers** & **Consumers** w/**SizedQueue**

```
qs = Array.new(n){ SizedQueue.new 1 }
```



```
producers = []  
n.times do |i|  
  thread = Thread.new do  
    ...  
    each_with_index do |args, j|  
      every_nth = j.modulo(n) == i  
      next unless every_nth  
      qs[ j.modulo(n) ].push( [args, j] )  
    end  
    qs[ i ].push( Forkoff.done )  
  end  
  producers << thread  
end
```

qs is a SizedQueue that holds one item. SizedQueue, which can be found in thread.rb in Ruby's stdlib, is a thread-safe data structure.

Forkoff begins by getting producer threads going and putting items to be processed in SizedQueues.


Forkoff: **Producers** & **Consumers** w/**SizedQueue**

```
consumers = []
n.times do |i|
  thread = Thread.new do
    loop do

      value = qs[i].pop
      break if value == Forkoff.done
      args, index = value
      r, w = IO.pipe
      pid = fork

      unless pid
        r.close
        ...
        result = block.call(*args)
        ...
        w.write( Marshal.dump( result ) )
        exit
      end

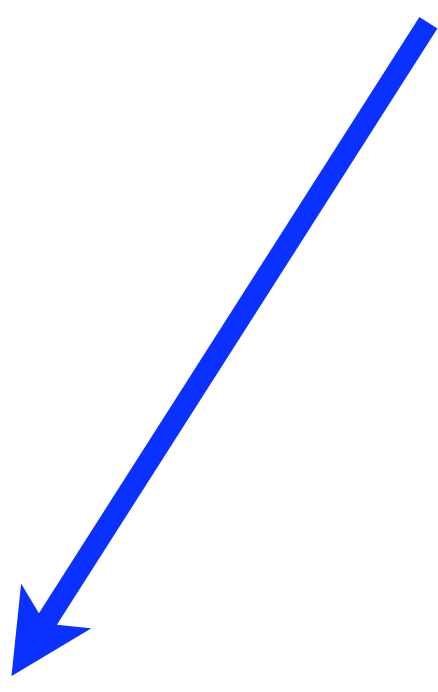
      w.close
      result = Marshal.load( r.read )
      results[i].push( [result, index] )
      Process.waitpid pid
    end
  end
  consumers << thread
end
```



The consumers threads will wait until something is ready for them to process. The way SizedQueue works, “pop” blocks until there’s something available in the queue.

Forkoff: **Producers** & **Consumers** w/**SizedQueue**

```
producers = []  
  
n.times do |i|  
  thread = Thread.new do  
    ...  
    each_with_index do |args, j|  
      every_nth = j.modulo(n) == i  
      next unless every_nth  
      qs[ j.modulo(n) ].push( [args, j] )  
    end  
    qs[ i ].push( Forkoff.done )  
  end  
  producers << thread  
end
```



When there's nothing more to pass to any producer thread, Forkoff.done will be passed to the queue.

Forkoff: Producers & Consumers w/SizedQueue

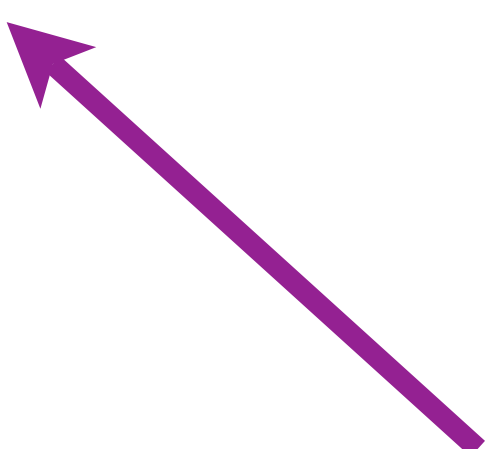
```
consumers = []
n.times do |i|
  thread = Thread.new do
    loop do
      value = qs[i].pop

      break if value == Forkoff.done

      args, index = value
      r, w = IO.pipe
      pid = fork

      unless pid
        r.close
        ...
        result = block.call(*args)
        ...
        w.write( Marshal.dump( result ) )
        exit
      end

      w.close
      result = Marshal.load( r.read )
      results[i].push( [result, index] )
      Process.waitpid pid
    end
  end
end
consumers << thread
end
```



The consumer threads will shut down one by one when they detect the “done” indicator, as the data supply dwindles.

Forkoff and peach are great as far as they go, but to really take advantage of multi-core technology -- features like figuring out how many cores are available on the library level, and even load balancing between cores, are important.

Intel's Threading Building Blocks (<http://www.threadingbuildingblocks.org/>) is an example of a library that provides that kind of functionality today. It can make routing decisions based on the current cache contents. TBB just added a feature that makes it possible to do things like designate a thread for actions that might block, like IO, while a computation is in progress.

The Omnibus concurrency library (<http://rubyforge.org/projects/concurrent/>) includes functions like peach, in addition to support for a number of different concurrency models. It's compatible with Ruby 1.8 and JRuby. It was created by MenTaLguY, who has done a lot of the concurrency design for both JRuby and Rubinius, and who wrote the fastthread library, which fixes memory leaks and improves performance for Ruby 1.8.5.

The next version of Omnibus will support some TBB-like features. MenTaLguY describes some of his plans for Omnibus in this blog entry: <http://moonbase.rydia.net/mental/blog/programming/the-future-of-the-omnibus>.

Using Omnibus Library Actors

```
require 'rubygems'
require 'concurrent/actors'

Actor = Concurrent::Actors::Actor

actor = Actor.spawn do
  loop do
    Actor.receive do |f|
      f.when( "fanmail" ) { send_autographed_picture }
      f.when( "applause" ) { bow }
    end
  end
end

actor << "applause"
```



DHH isn't really setting up a dichotomy between Threads and Processes in the multi-core blog entry we looked at. He's really asserting that there are alternatives to threads.

I'd like to show you an example of a concurrency model that's very different from threads, but I'll show how it can be used for some of the same functions. It's the Actor model, as it is implemented in Omnibus. Actors are lightweight concurrency primitives with a built-in message-passing API.

In brief, an Actor receives messages in its mailbox, where they can be filtered. A call to receive blocks, effectively putting the actor to sleep until there is a message.

Here's a trivial toy example. The “<<” is used to send a message to an Actor. In this example the Actor will “bow” upon receiving “applause”. Sending “fanmail,” would invoke this Actor's “send_autographed_picture” method. (Thanks to MenTaLguY for helping me make this into a runnable example).

In the current version of Omnibus, each Actor has its own thread, but in future versions of Omnibus, there will be a pool of worker threads for handling Actor tasks.

Rubinius

+



```
pool = (0...POOL_SIZE).map {  
  Rubinius::VM.spawn "rbx_mongrel.rb"  
}  
loop do  
  
  # server is a regular TCPServer  
  socket = server.accept  
  vm = pool.shift  
  
  # send the file descriptor num to the other vm  
  vm << socket.fileno  
  pool.push vm  
end
```

What could you use Actors for in real life?

Here's the Rubinius code I showed you before to give you an idea of what spawning Mongrels in separate VMs would look like. The “<<” is used to send a message to a VM because the underlying protocol for inter-VM communication is based on Actors.

Rubinius

+



```
pool = (0...POOL_SIZE).map {  
  Rubinius::VM.spawn "rbx_mongrel.rb"  
}  
loop do
```

```
  # server i  
  socket = s  
  vm = pool.
```

```
  # send the  
  vm << sock  
  pool.push  
end
```

```
def run  
  while true  
    # client = @socket.accept  
    fd = Rubinius::VM.get_message  
    # create a local IO object for this fd  
    client = IO.for_fd(fd)  
    thread = Thread.new(client)  
      {|c| process_client(c) }  
    thread[:started_on] = Time.now  
    sleep @throttle if @throttle > 0  
    graceful_shutdown  
  end  
end
```

And here's another look at the the modified Mongrel runner, where a VM receives the socket file descriptor.

Revactor

Network programming for a concurrent world



```
def run
  ...
  while true
    client = @socket.accept
    ...
    actor = Actor.spawn client,
      &method(:process_client)
    actor[:started_on] = Time.now
    @workers.add(thread)
    sleep @throttle if @throttle > 0
  end
  graceful_shutdown
  ...
end
```

Here's another bit of practical Actor code. It's an abbreviated version of code from the samples that come with Revactor, Tony Arcieri's Actor library for Ruby 1.9 (<http://revactor.org/>). It's a working Mongrel example that uses actors for processing requests, instead of threads.

Me June 06, 2007 11:28

[About](#)

[Gallery](#)

[Company](#)

[Girl](#)

Projects

[Ruby on](#)

[Rails](#)

[Basecamp](#)

[Highrise](#)

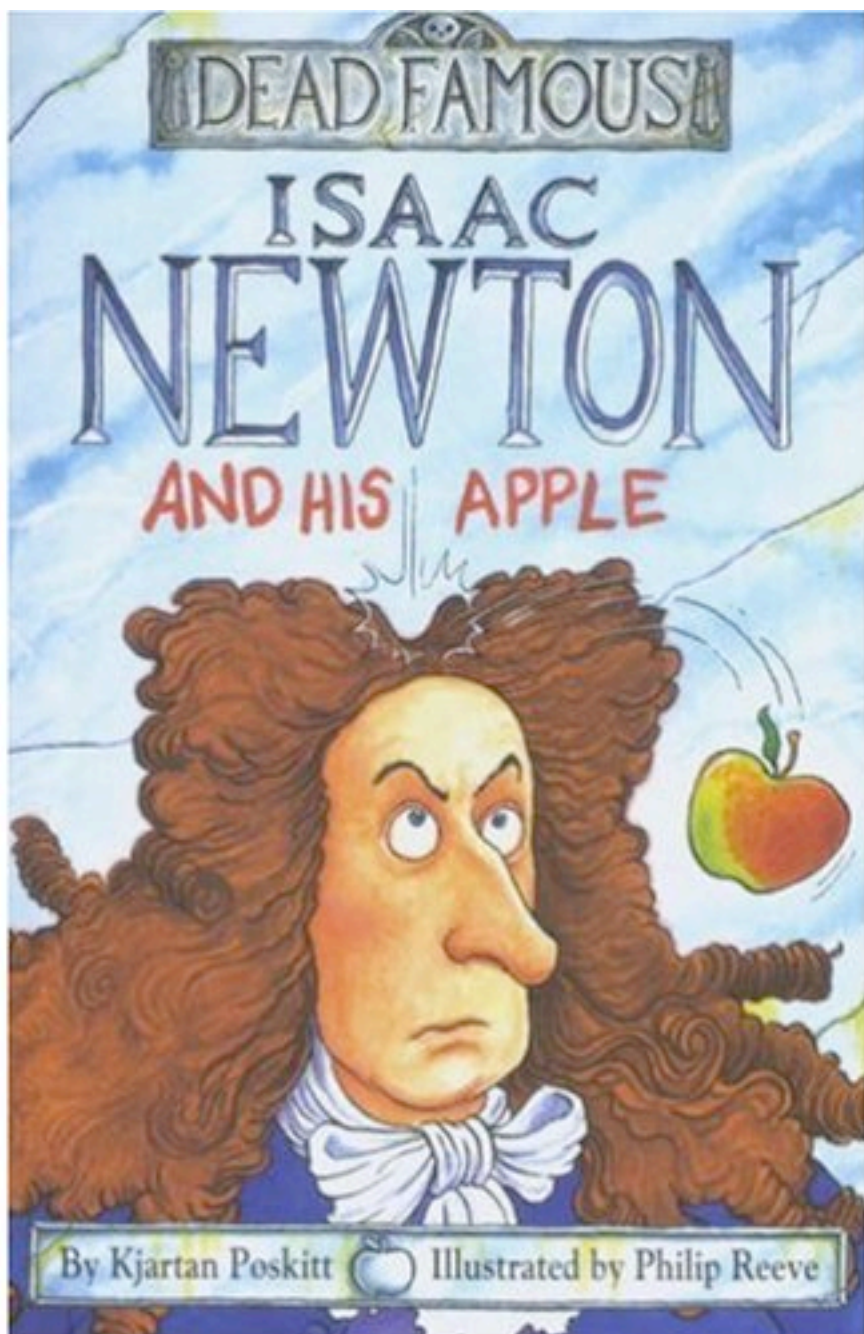
[Backpack](#)

[Campfire](#)

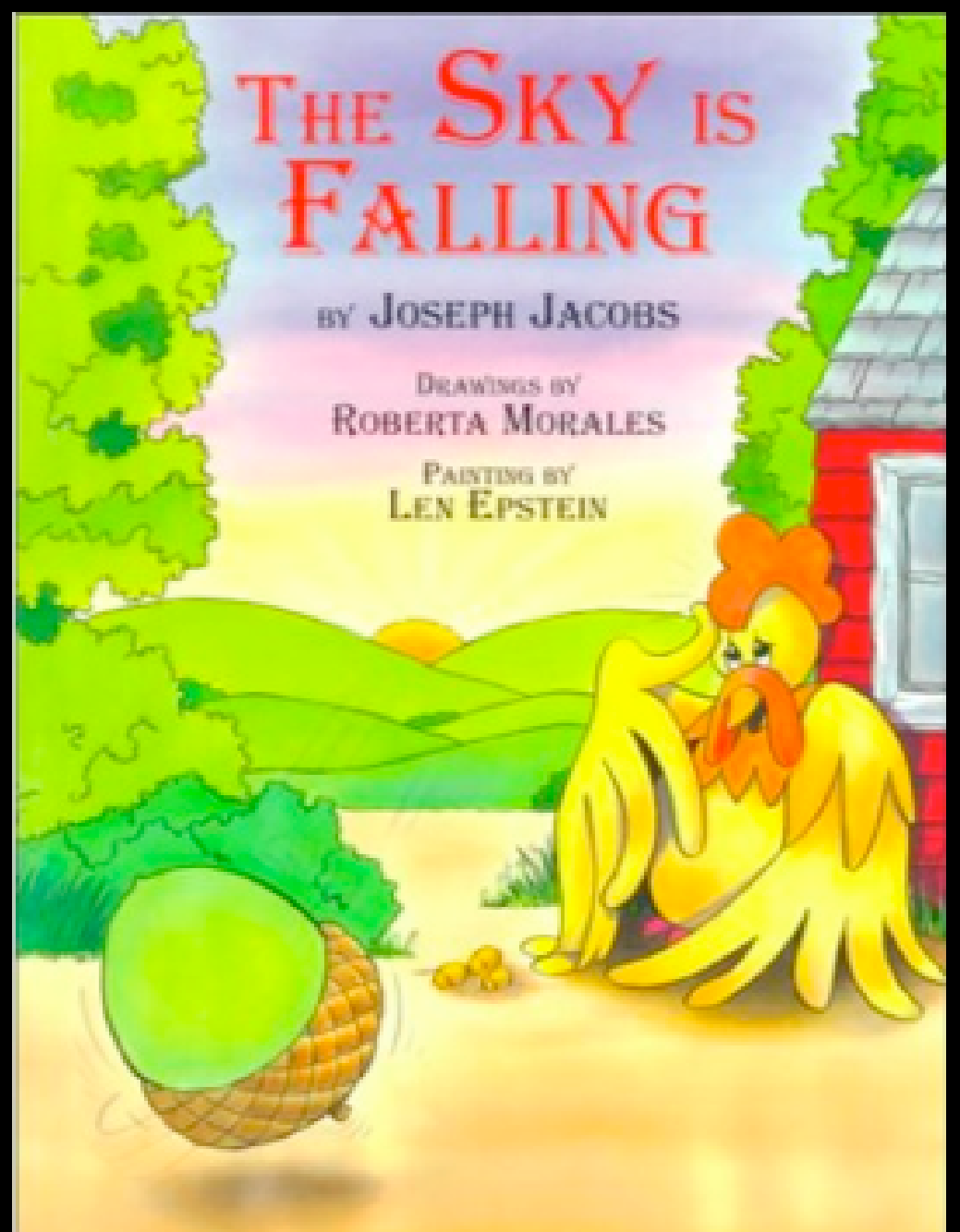
Multi-core Hysteria and the Thread Confusion

[Advances in multicore technology have] caused quite a few folks to pontificate that **the sky is falling for Rails** because we're not big on using threads. It isn't.

In wrapping up, I'd like to go back to David's blog entry, and its reference to the story of Chicken Little, who immediately started running around telling everyone the sky is falling after an acorn fell on his head.



http://www.amazon.com/Isaac-Newton-His-Apple-Famous/dp/0590114069/ref=sr_1_1?ie=UTF8&s=books&qid=1211454594&sr=1-1



http://www.amazon.com/Sky-Falling-Joseph-Jacobs/dp/0016859092/ref=sr_1_36?ie=UTF8&s=books&qid=1211454389&sr=8-36

Speaking of parallelism, the famous story of Isaac Newton and the apple is essentially the same story as the story of Chicken Little, but in a parallel universe.

Wikipedia offers this in one definition of parallel universes: “there are an infinite number of universes and that everything that could possibly happen in our universe (but doesn't) does happen in another.”

What if instead of getting hysterical, Chicken Little had come up with the laws of natural physics? The story of Chicken Little would be quite a different story, then!

These books represent two very different paths. As the chant that Ruby was not equipped to deliver in a multicore environment grew louder, the Ruby community could have taken the path of around yelling “The sky is falling”.

Developers are exploring interesting ideas like Erlang–Ruby bridges and Ruby–based implementations of map–reduce, and the implementors of the different VMs are working on innovative ways to not only improve Ruby’s concurrency model, but to do groundbreaking work.